Functional and Operational Solutions for Safety Reconfigurable Embedded Control Systems

Atef Gharbi, Mohamed Khalgui and Mohammad Ayoub Khan

Abstract The chapter deals with run-time automatic reconfigurations of distributed embedded control systems following component-based approaches. We classify reconfiguration scenarios into four forms: (1) additions-removals of components, (2) modifications of their compositions, (3) modifications of implementations, and finally (4) simple modifications of data. We define a new multi-agent architecture for reconfigurable systems where a Reconfiguration Agent which is modelled by nested state machines is affected to each device of the execution environment to apply local reconfigurations, and a Coordination Agent is proposed for any coordination between devices in order to guarantee safe and coherent distributed reconfigurations. We propose technical solutions to implement the whole agent-based architecture, by defining UML meta-models for agents. In the execution scheme, a task is assumed to be a set of components having some properties independently from any real-time operating system. To guarantee safety reconfigurations of tasks at run-time, we define service and reconfiguration processes for tasks and use the semaphore concept to ensure safety mutual exclusions. We apply the priority ceiling protocol as a method to ensure the scheduling between periodic tasks with precedence and mutual exclusion constraints.

A. Gharbi (✉) · M. Khalgui
INSAT, Tunis, Tunisia
e-mail: atef.elgharbi@gmail.com

M. Khalgui
e-mail: khalgui.mohamed@gmail.com

M. A. Khan
Sharda University, Gr. Noida, India
e-mail: ayoub@ieee.org

1 Introduction

Nowadays, the new generations of distributed embedded control systems are more and more sophisticated since they require new forms of properties such as reconfigurability, reusability, agility, adaptability and fault-tolerance. The first three properties are offered by new advanced component-based technologies, whereas the last two properties are ensured by new technical solutions such as multi-agent architectures.

New generations of component-based technologies have recently gained popularity in industrial software engineering since it is possible to reuse already developed and deployed software components from rich libraries. A Control Component is a software unit owning data of the functional scheme of the system. This advantage reduces the time to market and allows minimizations of the design complexity by supporting the system's software modularity. This chapter deals with run-time automatic reconfigurations of component-based applications by using multi-agent solutions. An agent is assumed to be a software unit allowing the control of the system as well as its environment before applying automatic reconfigurations. The reasons for which reconfigurations may be taken are classified into two categories [33]: (1) corrective reasons: if there is one component which is misbehaving, then it is automatically substituted by a new one which is assumed to run correctly. The new component is supposed to have the same functionalities as the old one. (2) Adaptive reasons: even the component-based application is running well, dynamic adaptations may be needed as a response to the new environment evolutions, in order to extend new functionalities or to improve some required functional properties. Dynamic reconfigurations can cover the following issues: (1) architecture level which means the set of components to be loaded in memory to constitute the implemented solution of the assumed system; (2) control level which means the compositions of components; (3) implementation level which means the behavior of components encoded by algorithms; and (4) data level which means the global values. We define a multi-agent architecture for reconfigurable embedded control systems where a Reconfiguration Agent is affected to each device of the execution environment to apply automatic reconfigurations of local components, and a Coordination Agent which is used for coordination between distributed Reconfiguration Agents in order to allow coherent distributed reconfigurations. The Coordination Agent is based on a coordination protocol using coordination matrices which define coherent simultaneous reconfigurations of distributed devices. We propose useful meta-models for Control Components and also for intelligent agents. These meta-models are used to implement adaptive embedded control systems. As we choose to apply dynamic scenarios, the system should run even during automatic reconfigurations, while preserving correct executions of functional tasks.

Given that Control Components are defined in general to run sequentially, this feature is inconvenient for real-time applications which typically handle several inputs and outputs in a too short time constraint. To meet performance and timing requirements, a real-time must be designed for concurrency. To do so, we define at the operational level some sequential program units called real-time tasks.

Thus, we define a real-time task as a set of Control Components having some real-time constraints. We characterize a task by a set of properties independently from any Real Time Operating System (RTOS). We define service processes as software processes for tasks to provide system's functionalities, and define reconfiguration processes as tasks to apply reconfiguration scenarios at run-time. In fact, service processes are functional tasks of components to be reconfigured by reconfiguration processes. To guarantee a correct and safety behavior of the system, we use semaphores to ensure the synchronization between processes. We apply the famous algorithm of synchronization between reader and writer processes such that executing a service is considered as a reader and reconfiguring a component is assumed to be a writer process. The proposed algorithm ensures that many service processes can be simultaneously executed, whereas reconfiguration processes must have exclusive access. We study in particular the scheduling of tasks through a Real Time Operating System. We apply the priority ceiling protocol proposed by Sha et al. [49] to avoid the problem of priority inversion as well as the deadlock between the different tasks. The priority ceiling protocol supposes that each semaphore is assigned a priority ceiling which is equal to the highest priority task using this semaphore. Any task is only allowed to enter its critical section if its assigned priority is higher than the priority ceilings of all semaphores currently locked by other tasks.

In this chapter, we continue our research by proposing an original implementation of this agent-based architecture. We assume that agent controls the plant to ensure the system running physically. The design and the implementation of such agent under Real-Time constraints are the scope of this study. The main contributions of this chapter are the following: (1) a complete study of Safety Reconfigurable Embedded Control Systems from the functional level (i.e. dynamic reconfiguration system with a multi-agent system) to the operational level (i.e. decomposition of the system into a set of tasks with time constraints); (2) a global definition of real-time task with its necessary parameters independently from any real-time operating system; (3) the scheduling of these real-time tasks considered as periodic tasks with precedence and mutual exclusion constraints. To our best of knowledge, there is no research works which deal with these different points together.

We present in Sect. 2 the state of art about dynamic reconfiguration. Section3 presents the benchmark production systems FESTO and EnAS that we follow as running examples in the chapter. We define in Sect. 4 a multi-agent architecture and the communication protocol to ensure safety in a distributed embedded control systems. Section 5 presents the real-time task model and studies the safety of its dynamic reconfiguration as well as the scheduling between the different tasks. We finally conclude the chapter in Sect. 6.

2 Dynamic Reconfiguration

The new generation of industrial control systems is addressing today new criteria as flexibility and agility [43, 48]. We distinguish two reconfiguration policies: *static* and *dynamic* policies such that static reconfigurations are applied off-line to apply changes before any system cold start [3], whereas dynamic reconfigurations are dynamically applied at run-time. Two cases exist in the last policy: manual reconfigurations applied by users [47] and automatic reconfigurations applied by intelligent agents [2]. We are interested in automatic reconfigurations of an agent-based embedded control system when hardware or software faults occur at run-time. The system is implemented by different complex networks of Control Components. In literature, there are various studies about dynamic reconfigurations applied to component-based applications. Each study has its strength and its weakness. In the article [35], the authors propose to block all nodes involved in transactions (considered as sets of interactions between components) to realize dynamic reconfigurations. This study has influenced many research works later. Any reconfiguration should respect the consistency propriety which is defined as sets of logical constraints. A major disadvantage of this approach is the necessity to stop all components involved in a transaction. In the article [4], problem of dynamic reconfigurations in CORBA is treated. The authors consider that consistency is related to Remote Procedure Call Integrity. To ensure this property, they propose to block the incoming before the outgoing links. However, the connection between components must be acyclic in order to be able to block connections in the right order. A dynamic reconfiguration language based on features [41] is proposed. The authors use the control language MANIFOLD where processes are considered as black boxes having ports of communication. In this case, the communication is anonymous. The processes having access to shared data are connected in cyclic manners to wait tokens that visit each one at turn (as in token ring). Although the novelty of this solution, there is a loss of time especially at waiting until receiving the token to access to the shared data or also to reconfigure the system. Another study [46] is proposed to apply dynamic updates on graphical components (for example button, graphical interface, . . .) in a .Net framework. To do so, the authors associate for each graphical component an appropriate running thread. The synchronization is ensured through the reader-writer-locks. The dynamic reconfiguration is based on blocking all involved connections. Due to rw-locks, this solution works only on local applications. In addition, they define [45] a new reconfiguration algorithm ReDAC (Reconfiguration of Distributed Application with Cyclic dependencies) ensuring dynamic reconfigurations in distributed systems to be based on running multi-threads. This algorithm is applied to capsules which are defined as groups of running components. As disadvantage, the proposed algorithm uses counter variables to count on-going method calls for threads which lead to consume further space memory and treatment time.

To our best of knowledge, there is no research works which treat the problem of dynamic software reconfigurations of component-based technology with semaphores. The novelty of this chapter is the study of dynamic reconfiguration with semaphore ensuring the following points: (1) blocking connections without blocking involved components; (2) safety and correctness of the proposed solution; (3) independence of any specific language; (4) verification of consistency (i.e. logical constraints) delegated to the software agent; (5) suitable for large-scale applications.

3 Benchmark Production Systems: FESTO and EnAS

We present two Benchmark Production Systems[1]: FESTO and EnAS available in the research laboratory at the Martin Luther University in Germany.

3.1 The FESTO System

The FESTO Benchmark Production System is a well-documented demonstrator used by many universities for research and education purposes, and it is used as a running example in the context of this chapter. FESTO is composed of three units: Distribution, Test and Processing units. The Distribution unit is composed of a pneumatic feeder and a converter to forward cylindrical work pieces from a stack to the testing unit which is composed of the detector, the tester and the elevator. This unit performs checks on work pieces for height, material type and color. Work pieces that successfully pass this check are forwarded to the rotating disk of the Processing unit, where the drilling of the work piece is performed. We assume in this research work two drilling machines $Drill\_machine1$ and $Drill\_machine2$ to drill pieces. The result of the drilling operation is next checked by the checking machine and the work piece is forwarded to another mechanical unit. In this research chapter, three production modes of FESTO are considered according to the rate of input pieces denoted by $number\_pieces$ into the system (i.e. ejected by the feeder).

- **Case 1**: **High production**. **If** $number\_pieces \geq Constant1$, **then** the two drilling machines are used at the same time in order to accelerate the production. In this case, the Distribution and the Testing units have to forward two successive pieces to the rotating disc before starting the drilling with $Drill\_machine1$ **AND** $Drill\_machine2$. For this production mode, the periodicity of input pieces is $p = 11$ s.
- **Case 2**: **Medium production**. **If** $Constant2 \leq number\_pieces < Constant1$, **then** we use $Drill\_machine1$ **OR** $Drill\_machine2$ to drill work pieces. For this production mode, the periodicity of input pieces is $p = 30$ s.
- **Case 3**: **Light production**. **If** $number\_pieces < Constant2$, **then** only the drilling machine $Drill\_machine1$ is used. For this production mode, the periodicity of input pieces is $p = 50$ s.

On the other hand, if one of the drilling machines is broken at run-time, then we have to only use the other one. In this case, we reduce the periodicity of input pieces to $p = 40$ s. The system is completely stopped in the worst case if the two drilling machines are broken.

[1] Detailed descriptions are available in the website: http://aut.informatik.uni-halle.de.

3.2 The EnAS System

The Benchmark Production System EnAS was designed as a prototype to demonstrate energy-antarcic actuator/sensor systems. For the sale of this contribution, we assume that it has the following behavior: it transports pieces from the production system (i.e. FESTO system) into storing units. The pieces in EnAS shall be placed inside tins to close with caps afterwards. Two different production strategies can be applied: we place in each tin one or two pieces according to production rates of pieces, tins and caps. We denote respectively by nb_{pieces}, $nb_{tins+caps}$ the production number of pieces and tins (as well as caps) per hour and by *Threshold* a variable (defined in user requirements) to choose the adequate production strategy. The EnAS system is mainly composed of a belt, two Jack stations (J_1 and J_2) and two Gripper stations (G_1 and G_2). The Jack stations place new produced pieces and close tins with caps, whereas the Gripper stations remove charged tins from the belt into storing units. Initially, the belt moves a particular pallet containing a tin and a cap into the first Jack station J_1.

According to production parameters, we distinguish two cases,

- **First production policy**: **If** ($nb_{pieces}/nb_{tins+caps} \leq Threshold$), **then** the Jack station J_1 places from the production station a new piece and closes the tin with the cap. In this case, the Gripper station G_1 removes the tin from the belt into the storing station St_1.
- **Second production policy**: **If** ($nb_{pieces}/nb_{tins+caps} > Threshold$), **then** the Jack station J_1 places just a piece in the tin which is moved thereafter into the second Jack station to place a second new piece. Once J_2 closes the tin with a cap, the belt moves the pallet into the Gripper station G_2 to remove the tin (with two pieces) into the second storing station St_2.

4 Multi-agent System

We define a multi-agent architecture for distributed safety systems. Each reconfiguration agent is affected in this architecture to a device of the execution environment to ensure Functional Safety. Nevertheless, the coordination between agents in this distributed architecture is inevitable because any individual decision may affect the performance of the others. To guarantee safe distributed reconfigurations, we define the concept of *Coordination Matrix* that defines correct reconfiguration scenarios to be applied simultaneously in distributed devices and we define the concept of *Coordination Agent* that handles coordination matrices to coordinate between distributed agents. We propose a communication protocol between agents to manage concurrent distributed reconfiguration scenarios.

The communication protocol between agents respects the different following points: (1) The Reconfiguration agents control the plant constituted by several physical processes. (2) At the beginning, all the Reconfiguration agents are assigned

Table 1 The agent characteristics

Agent type	Percepts	Actions	Goals	Environment
Reconfiguration agent	Something needs an intervention	Reconfigure the plant	Safe state	Physical plant
Coordination agent	Reconfiguration request	Contact the other agents	Coordination between agents	The whole system

a specific reconfiguration. (3) The Reconfiguration agent controlling the system can not apply more than one reconfiguration at any time. (4) The Reconfiguration agent decides to apply a new reconfiguration if some conditions are verified. (5) The Reconfiguration may be applied in a local system (in this case, only the associated Reconfiguration agent is concerned) or in a distributed system (in this case, many Reconfiguration agents have to coordinate together to put the whole system in a safe state). (6) The Reconfiguration agent does not know if the other agents will cooperate to put the system into safe state. (7) At the reception of a reconfiguration request, the agent chooses one action from the available possibilities (accept or refuse). The Reconfiguration agent may refuse the request if it is not possible to apply this new reconfiguration. (8) An agent is called cooperative if it always accepts the reconfiguration request. An agent is called selfish if it always refuses the new reconfiguration.

Before introducing the communication protocol, we begin with presenting a Reconfiguration Agent as well as the coordination agent. To resume the characteristics of each one, the Table 1 presents the main information.

4.1 Software Architecture of Reconfiguration Agents

We propose an agent-based architecture to control embedded systems at run-time. The agent checks the environment's evolution and reacts when new events occur by adding, removing or updating Control Components of the system. To describe the dynamic behavior of an intelligent agent that dynamically controls the plant, we use nested state machines in which states correspond to finite state machines. A finite state machine can be defined as a state machine whose states, inputs and outputs are enumerated. The nested state machine is represented as the following:

$$\text{NSM} = (SM_1, SM_2, \ldots, SM_n)$$

Each state machine (SM_i) is a graph of states and transitions. A state machine treats the several events that may occur by detecting them and responding to each one appropriately. We define a state machine as the following:

$$SM_i = (S_i, S_{i0}, I_i, O_i, \mathit{Pre\text{-}cond}_i, \mathit{Post\text{-}cond}_i, t_i)$$

- $S_i = \{s_{i1}, \ldots, s_{ip}\}$: the states;
- S_{i0} the initial state;
- $I_i = \{I_{i1}, \ldots, I_{im}\}$: the input events;
- $O_i = \{O_{i1}, \ldots, O_{ik}\}$: the ouput events;
- $Pre\text{-}cond_i$: the set of conditions to be verified before the activation of a state;
- $Post\text{-}cond_i$: the set of conditions to be verified once a state is activated;
- $t_i : S_i \times I_i \rightarrow S_i$: the transition function.

We propose a conceptual model for a nested state machine in Fig. 1 where we define the classes *Nested State Machine*, *State machine*, *State*, *Transition*, *Event* and *Condition*. The *Nested State Machine* class contains a certain number of *State machine* classes. This relation is represented by a composition. The *Transition* class is double linked to the *State* class because a transition is considered as an association between two states. Each transition has an event that is considered as a trigger to fire it and a set of conditions to be verified. This association between the *Transition* class and *Event* and *Condition* classes exists and is modeled by the aggregation relation.

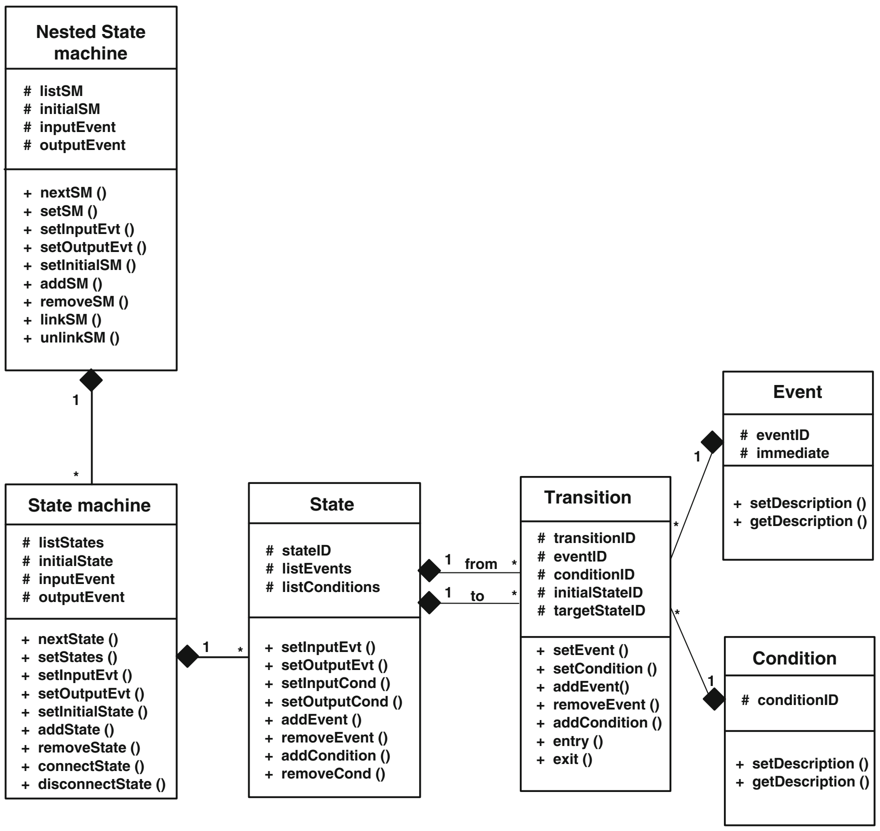

Fig. 1 The Meta-model nested state machine

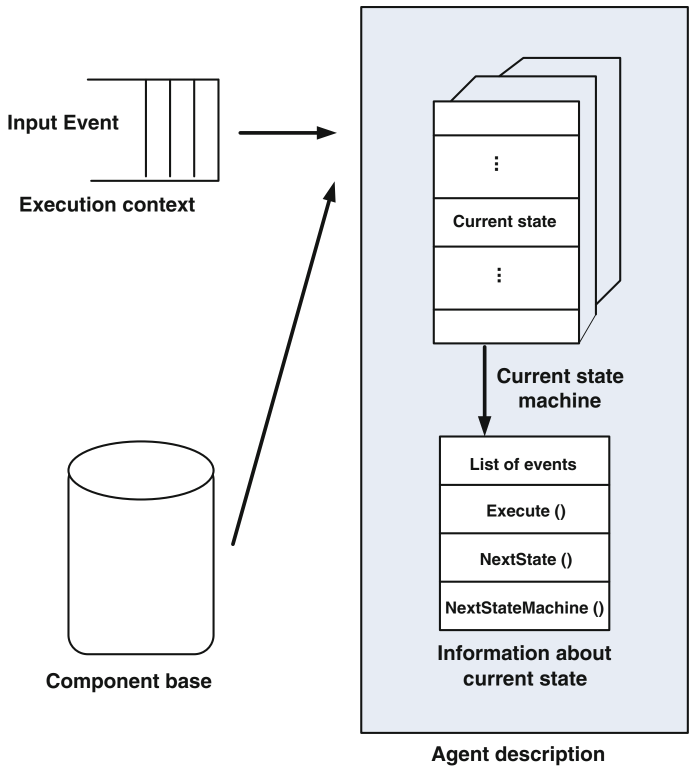

Fig. 2 The internal agent behavior

We propose a generic architecture for intelligent agents depicted in Fig. 2. This architecture consists of the following parts: (1) the Event Queue to save different input events that may take place in the system, (2) the intelligent software agent that reads an input event from the Event Queue and reacts as soon as possible, (3) the set of state machines such that each one is composed of a set of states, (4) each state represents a specific information about the system. The agent, based on nested state machines, determines the new system's state to execute according to event inputs and also conditions to be satisfied. This solution has the following characteristics: (1) The control agent design is general enough to cope with various kinds of embedded-software based-component application. Therefore, the agent is uncoupled from the application and from its Control Components. (2) The agent is independent of nested state machines: it permits to change the structure of nested state machines (add state machines, change connections, change input events, and so on) without having to change the implementation of the agent. This ensures that the agent continues to work correctly even in case of modification of state machines. (3) The agent is not supposed to know components that it has to add or remove in a reconfiguration case.

In the following algorithm, the symbol Q is an event queue which holds incoming event instances, ev refers to an event input, S_i represents a State Machine, and $s_{i,j}$ a state related to a State Machine S_i. The internal behavior of the agent is defined as follow:

1. the agent reads the first event ev from the queue Q;
2. searches from the top to the bottom in the different state machines;
3. within the state machine SM_i, the agent verifies if ev is considered as an event input to the current state $s_{i,j}$ (i.e. ev $\in$ I related to $s_{i,j}$). In this case, the agent searches the states considered as successor for the state $s_{i,j}$ (states in the same state machine SM_i or in another state machine SM_l);
4. the agent executes the operations related to the different states;
5. repeats the same steps (1–4) until no more event exists in the queue to be treated.

Algorithm 1: GenericBehavior

```
begin
while ($Q.length() > 0$) do
    ev $\leftarrow$ $Q.Head()$
    For each state machine $SM_i$ do
        $s_{i,j} \leftarrow currentState_i$
        If ev $\in I(s_{i,j})$ then
            For each state $s_{i,k} \in next(s_{i,j})$
             such that $s_{i,k}$ related to $S_i$ do
                If execute($s_{i,k}$) then
                    $currentState_i \leftarrow s_{i,k}$
                    break
                end if
            end for
            For each state $s_{l,k} \in next(s_{i,j})$
             such that $s_{l,k}$ related to $S_l$ do
                If execute($s_{l,k}$) then
                    $currentState_l \leftarrow s_{l,k}$
                    break
                end if
            end for
        end if
    end for
end while
end.
```

First of all, the agent evaluates the pre-condition of the state $s_{i,j}$. If it is false, then the agent exits, Else the agent determines the list of Control Components concerned by this reconfiguration, before applies the required reconfiguration for each one. Finally, it evaluates the post-condition of the state $s_{i,j}$ and generates errors whenever it is false.

```
    **Function** execute($s_{i,j}$) : boolean
begin
    If $\neg s_{i,j}.PreCondition$ then
        return false
    else
        listCC ← getInfo($s_{i,j}$.info)
        For each CC ∈ listCC do
            CC.reconfigure()
        end for
        If $\neg s_{i,j}.PostCondition$ then
            Generate error
        end if
        return true
    end if
end.
```

4.2 Communication Protocol

To guarantee safe distributed reconfigurations, we define the concept of *Coordination Matrix* that defines correct reconfiguration scenarios to be applied simultaneously in distributed devices and we define the concept of *Coordination Agent* that handles coordination matrices to coordinate between distributed agents.

Let Sys be a distributed safe system of n devices, and let $Ag_1, \ldots, Ag_n$ be n agents to handle automatic distributed reconfiguration scenarios of these devices. We denote in the following by $Reconfiguration^a_{i_a,j_a,k_a,h_a}$ a reconfiguration scenario applied by Ag_a ($a \in [1, n]$) as follows: (1) the corresponding ASM state machine is in the state ASM_{i_a}. Let $cond^a_{i_a}$ be the set of conditions to reach this state, (2) the CSM state machine is in the state CSM_{i_a,j_a}. Let $cond^a_{j_a}$ be the set of conditions to reach this state, (3) the DSM state machine is in the state DSM_{k_a,h_a}. Let $cond^a_{k_a,h_a}$ be the set of conditions to reach this state. To handle coherent distributed reconfigurations that guarantee safe behaviors of the whole system Sys, we define the concept of *Coordination Matrix* of size (n, 4) that defines coherent scenarios to be simultaneously applied by different agents. Let CM be such a matrix that we characterize as follows: each line a ($a \in [1, n]$) corresponds to a reconfiguration scenario $Reconfiguration^a_{i_a,j_a,k_a,h_a}$ to be applied by Ag_a as follows:

$$CM[a, 1] = i_a; \quad CM[a, 2] = j_a; \quad CM[a, 3] = k_a; \quad CM[a, 4] = h_a$$

According to this definition: **If** an agent Ag_a applies the reconfiguration scenario $Reconfiguration^a{}_{CM[a,1],CM[a,2],CM[a,3],CM[a,4]}$, **Then** each other agent Ag_b ($b \in [1, n]\backslash\{a\}$) has to apply the scenario $Reconfiguration^b{}_{CM[b,1],CM[b,2],CM[b,3],CM[b,4]}$ (Fig. 3). We denote in the following by *idle agent* each agent Ag_b ($b \in [1, n]$) which is not required to apply any reconfiguration when others perform scenarios defined in CM. In this case:

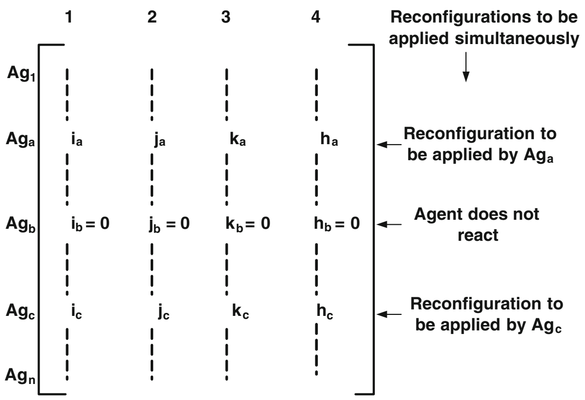
1
2
3
4
Reconfigurations to be applied simultaneously
Ag1
Aga
Agb
Agc
Agn
ia
ja
ka
ha
ib = 0
jb = 0
kb = 0
hb = 0
ic
jc
kc
hc
Reconfiguration to be applied by Aga
Agent does not react
Reconfiguration to be applied by Agc

Fig. 3 The coordination matrix

$$CM[b,1] = CM[b,2] = CM[b,3] = CM[b,4] = 0$$
$$cond^{a}_{CM[a,1]} = cond^{a}_{CM[a,2]} = cond^{a}_{CM[a,3],CM[a,4]} = True$$

We propose a communication protocol between agents to manage concurrent distributed reconfiguration scenarios. We guarantee a coherent behavior of the whole distributed system by defining a *Coordination Agent* (denoted by $CA(\xi(Sys))$) which handles the Coordination Matrices of $\xi(Sys)$ to control the rest of agents (i.e. Ag_a, $a \in [1,n]$) as follows:

- When a particular agent Ag_a ($a \in [1,n]$) should apply a reconfiguration scenario $Reconfiguration^{a}_{i_a,j_a,k_a,h_a}$ (i.e. under well-defined conditions), it sends the following request to $CA(\xi(Sys))$ to obtain its authorization:

$$request(Ag_a, CA(\xi(Sys)), Reconfiguration^{a}_{i_a,j_a,k_a,h_a}).$$

- When $CA(\xi(Sys))$ receives this request that corresponds to a particular coordination matrix $CM \in \xi(Sys)$ and if CM has the highest priority between all matrices of $Concur(CM) \cup \{CM\}$, then $CA(\xi(Sys))$ informs the agents that have simultaneously to react with Ag_a as defined in CM. The following information is sent from $CA(\xi(Sys))$:
  For each $Ag_b, b \in [1,n] \setminus \{a\}$ and $CM[b,i] \neq 0, \forall i \in [1,4]$: *reconfiguration* $(CA(\xi(Sys)), Ag_b, Reconfiguration^{b}_{CM[b,1],CM[b,2],CM[b,3],CM[b,4]})$
- According to well-defined conditions in the device of each Ag_b, the $CA(\xi(Sys))$ request can be accepted or refused by sending one of the following answers:
  - If $cond^{b}_{i_b} = cond^{b}_{j_b} = cond^{b}_{k_b,h_b} =$ True
    then the following reply is sent from Ag_b to $CA(\xi(Sys))$: *possible_reconfig* $(Ag_b, CA(\xi(Sys)), Reconfiguration^{b}_{CM[b,1],CM[b,2],CM[b,3],CM[b,4]})$.
  - Else the following reply is sent from Ag_b to $CA(\xi(Sys))$: *not_possible_reconfig* $(Ag_b, CA(\xi(Sys)), Reconfiguration^{b}_{CM[b,1],CM[b,2],CM[b,3],CM[b,4]})$.

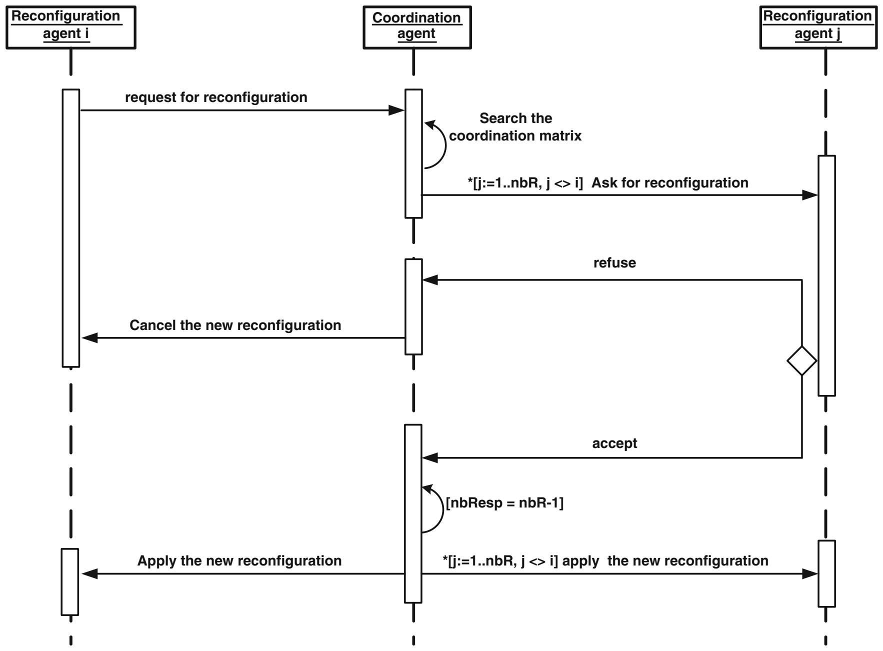
Reconfiguration agent i
Coordination agent
Reconfiguration agent j
request for reconfiguration
Search the coordination matrix
*[j:=1..nbR, j <> i] Ask for reconfiguration
refuse
Cancel the new reconfiguration
accept
[nbResp = nbR-1]
Apply the new reconfiguration
*[j:=1..nbR, j <> i] apply the new reconfiguration

Fig. 4 The communication scenario

- If $CA(\xi(Sys))$ receives positive answers from all agents, then it authorizes reconfigurations in the concerned devices: For each Ag_b, $b \in [1, n]$ and $CM[b, i] \neq 0$, $\forall i \in [1, 4]$, *apply* ($Reconfiguration^b_{CM[b,1],CM[b,2],CM[b,3],CM[b,4]}$) in $device_b$. Else If $CA(\xi(Sys))$ receives a negative answer from a particular agent, then
  - If the reconfiguration scenario $Reconfiguration^a_{i_a,j_a,k_a,h_a}$ allows optimizations of the whole system behavior, then $CA(\xi(Sys))$ refuses the request of Ag_a by sending the following reply: $refused\_reconfiguration(CA(\xi(Sys)), Ag_a, Reconfiguration^a_{CM[a,1],CM[a,2],CM[a,3],CM[a,4]}))$.

When a Reconfiguration Agent (denoted by RA_i) needs to apply a new reconfiguration, it sends a request to the Coordination Agent. The Coordination Agent asks all the known Reconfiguration Agents (denoted by RA_j, $\forall j \in [1..NbR], j <> i$ where NbR represents the number of Reconfiguration Agents) if it is possible to apply the new reconfiguration introduced as parameter. The Reconfiguration Agent (RA_j) studies this proposition and sends its response which may be accept or refuse the new reconfiguration (depending on its related state). Whenever the Coordination Agent receives positive responses from all the Reconfiguration Agents ($RA_j \forall j \in [1..NbR], j <> i$) (i.e. the number of positives answers is equal to $NbR-1$), then it decides to apply the new reconfiguration for all Reconfiguration Agents RA_j ($\forall j \in [1..NbR]$) by sending a confirmation message. Whenever the Coordination Agent receives only one negative response from a Reconfiguration Agents

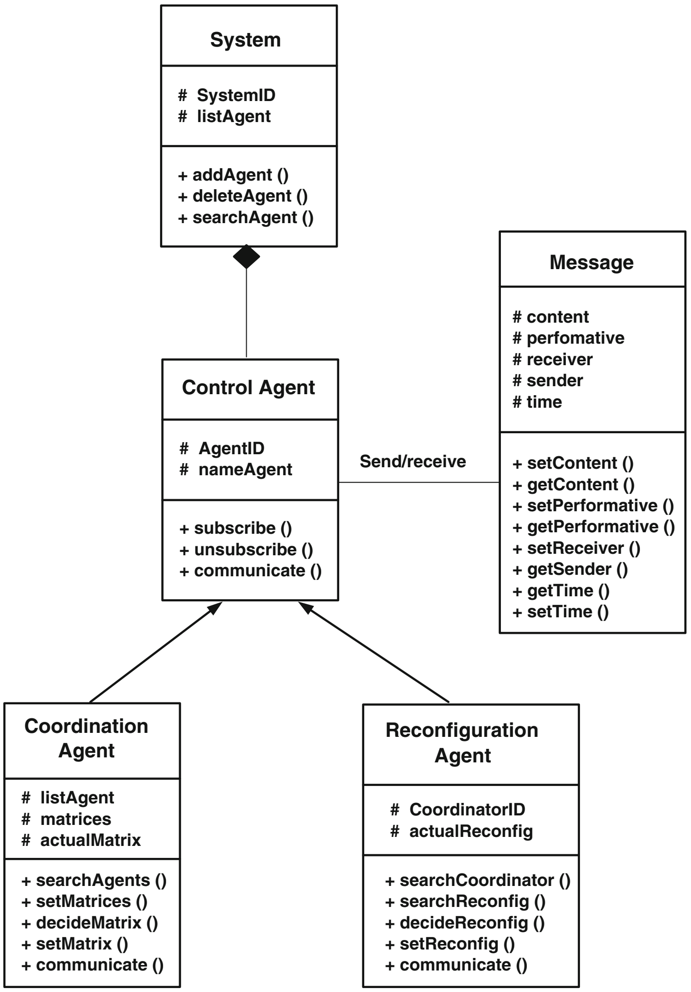

Fig. 5 The agent-based control in a distributed system

($RA_j, j \in [1..NbR], j <> i$), it decides to cancel this reconfiguration and informs the corresponding agent by its decision (i.e RA_i). Figure 4 depicts the interaction between Reconfiguration and Coordination agents to ensure dynamic reconfiguration in a distributed system.

Before sending or receiving a message, the Reconfiguration Agent searches the Coordination Agent with the method *searchCoordinator*(). The Coordination Agent in its turn searches also the list of Reconfiguration Agents with the method *searchAgents*().

The method *receive*() used by both Coordination Agent and Reconfiguration Agent permits to receive a message sent by another agent. Whenever *receive*() is invoked through *CA_Communicate*() and *RA_Communicate*() methods, if the agent does not receive a message, it is blocked (but without blocking the other activities of the same agent).

A message is defined by the following data: (1) *content*: the subject of the message (such as the reconfiguration to be applied); (2) *performative*: the performative indicates what the sender wants to achieve (for example ACCEPT, REFUSE, CANCEL, CONFIRM); (3) *time*: it is necessary to treat messages ordered by time; (4) *sender*: the agent emitting the message; and (5) *receiver*: the agent receiving the message. Figure 5 depicts the different classes such as *ControlAgent*, *CoordinationAgent*, *ReconfigurationAgent*, *Message* and *System*.

In the following, we present the *Communicate* method defined for both Reconfiguration and Coordination Agent. The *CA_Communicate* method defined for the Coordination Agent has as variables: (1) *i* representing the reconfiguration agent which initiates the request of reconfiguration; (2) *j* which corresponds to the reconfiguration agent receiving the request of reconfiguration from the coordination agent; (3) *NbR* which represents the total number of reconfiguration agents; (4) *NbResp* considered as the current number of responses approving the new reconfiguration by the reconfiguration agents; (5) *matrix* representing the new matrix to be applied if all the reconfiguration agents accept.

Algorithm *CA_Communicate*()

```
begin
switch (step)
case 0:
// Wait a request from a Reconfiguration Agent
    reply ← receive();
    if (reply != null)
        if (reply.getPerformative() = REQUEST)
            i ← reply.getSender();
            Matrix ← decideMatrix(reply.getContent());
            step++;
    else
        block();
    break;

    case 1:
// Send the proposition to all Reconfiguration Agents
    for j = 1 to NbR do
        if (j <> i)
            msg.addReceiver(reconfigurationAgents[j]);
            msg.setContent(Matrix[j]);
            msg.setPerformative(PROPOSE);
            msg.setTime(currentTime());
            send(msg);
    step++;
    break;
    case 2:
// Receive all accept/refusals from Reconfiguration Agents      reply ← receive();
    if (reply != null)
        if (reply.getPerformative() = ACCEPT)
            nbResp++;
            if (nbResp = nbR-1)
                step++;
        else
            if (reply.getPerformative() = REFUSE)
                step ← 4;
    else
        block();
    break;

    case 3:
// Send accept response to all Reconfiguration Agents
    for j = 1 to NbR do
        msg.addReceiver(reconfigurationAgents[j]);
        msg.setPerformative(CONFIRM);
        msg.setTime(currentTime());
        msg.setContent(Matrix[j]);
        send(msg);
        setMatrix(Matrix);
    step ← 0;
    break;

    case 4:
// Send refuse response to the Reconfiguration Agent i
    msg.addReceiver(reconfigurationAgents[i]);
    msg.setPerformative(CANCEL);
    msg.setTime(currentTime());
    send(msg);
    step ← 0;
    break;
end
```

The *RA_Communicate* method defines the Reconfiguration Agent behavior as follows: (1) whenever the Reconfiguration Agent receives a request to apply a new reconfiguration by the Coordination Agent, it evaluates this proposition and decides whether to accept or to refuse it. The Reconfiguration Agent sends its response. (2) whenever the Reconfiguration Agent receives a confirmation to apply the new reconfiguration from the Coordination Agent, then it applies it.

Algorithm ***RA_Communicate*()**

```
begin
switch (step)
case 0:
// Wait a request from a Coordination Agent
```

```
    reply ← receive();
    **if** (reply != null)
        **if** (reply.getPerformative() = REQUEST)
            newReconfig ← reply.getContent();
            response.setReceiver(CoordinatorID);
            **if** (decideReconfig(newReconfig))
                response.setPerformative(ACCEPT);
            **else**
                response.setPerformative(REFUSE);
            send(response)
            step++;
    **else**
        block();
    **break**;

    **case** 1:
// Wait the response from a Coordination Agent
    reply ← receive();
    **if** (reply != null)
        **if** (reply.getPerformative() = CONFIRM)
            setReconfig(newReconfig);
        step ← 0;
    **break**;
**end**
```

We developed a complete tool “ProtocolReconf”, to verify the communication protocol. The tool “ProtocolReconf” offers the possibility to create the Reconfiguration and Coordination Agents by introducing the necessary parameters. It is required to define the different scenarios that the Reconfiguration Agent can support so that when a modification occurs in the system, it should look for the convenient reconfiguration. For the Coordination Agent, it is necessary to define the set of Coordination Matrices to apply to the whole system [21].

5 Real-Time Task: Definition, Dynamic Reconfiguration and Scheduling

In this section, we present a Real-Time Task as a general concept independently from any real-time operating system, its dynamic reconfiguration, the scheduling between several tasks and the implementation in a specific real-time operating system (which is RTLinux).

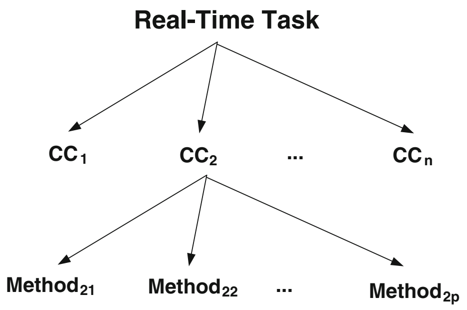

Fig. 6 Real time task

5.1 Real Time Task Definition

A real time task is considered as a process (or a thread depending on the Operating System) having its own data (such as registers, stack, ...) which is in competition with other tasks to have the processor execution. A task is handled by a Real-Time Operating System (RTOS) which is a system satisfying explicitly response-time constraints by supporting a scheduling method that guarantees response time especially to critical tasks.

In this paragraph, we aim to present a real-time task as a general concept independently from any real-time operating system.

To be independent from any Real-Time Operating System and to be related to our research work, we define a task τ_i as a sequence of Control Components, where a Control Component is ready when its preceding Control Component completes its execution. $\tau_{i,j}$ denotes the j-th Control Component of τ_i (Fig. 6). Thus, our application consists of a set of periodic tasks $\tau = (\tau_1, \tau_2, \ldots, \tau_n)$. All the tasks are considered as periodic this is not a limitation since non-periodic task can be handled by introducing a periodic server.

Running Example. *In the FESTO Benchmark Production System, the tasks* τ_1 *to* τ_9 *execute the following functions:*

- (τ_1) *Feeder pushes out cylinder and moves backward/back;*
- (τ_2) *Converter pneumatic sucker moves right/left;*
- (τ_3) *Detection Module detects workpiece, height, color and material;*
- (τ_4) *Shift out cylinder moves backward/forward;*
- (τ_5) *Elevator elevating cylinder moves down/up;*
- (τ_6) *Rotating disc workpiece present in position and rotary indexing table has finished a 90 rotation;*
- (τ_7) *Driller 1 machine drills workpiece;*
- (τ_8) *Driller 2 machine drills workpiece;*
- (τ_9) *WarehouseCylinder removes piece from table.*

In the following paragraphs, we introduce the meta-model of a task. We study also the dynamic reconfiguration of tasks. After that, we introduce the task scheduling.

Finally, we present the task implementation within RTLinux as a Real-Time Operating System.

5.2 A Meta-model Task

In this chapter, we extend the work presented in [42] by studying both a task and a scheduler in a general real-time operating system where each task is characterized by:

identifer: each task τ_i has a name and an identifier.

temporal properties: each task τ_i is described by a deadline D_i (which corresponds to the maximal delay allowed between the release and the completion of any instance of the task), a period T_i, a worst-case execution time C_i. It is released every T_i seconds and must be able to consume at most C_i seconds of CPU time before reaching its deadline D_i seconds after release ($C_i \leq D_i \leq T_i$). We assume that these attributes are known, and given as constants (Table 2).

constraints: resources specification ρ_i, precedence constraints and/or QoS properties to be verified.

state: A Real-Time Operating System implements a finite state machine for each task and ensures its transition. The state of a task may be in one of the following possible states Ready, Running, Blocked or Terminated. Every task is in one of a few different states at any given time:

Ready The task is ready to run but waits for allocation of the processor. The scheduler decides which ready task will be executed next based on priority criterion (i.e. the task having the highest priority will be assigned to the processor).

Blocked A task cannot continue execution because it has to wait (there are many reasons such that waiting for event, waiting on semaphore or a simple delay).

Running In the running state, the processor is assigned to the task, so that its instructions can be executed. Only one task can be in this state at any time, while all the other tasks can be simultaneously in other states.

Terminated When a task terminates its execution, the task allocator deletes it and releases the resources taken by this task (Fig. 7).

priority: each task is assigned a priority value which may be used in the scheduling.

$$\tau_i = (D_i; C_i; T_i; I_i; O_i; \rho_i; (CC_i^1, \ldots, CC_i^{n_i}));$$

- a deadline D_i;
- an execution time C_i;
- a period T_i;
- a set of inputs I_i;
- a set of outputs O_i;

Table 2 A task set example

Task	Comp. time C_i	Period T_i	Deadline D_i
τ_1	20	70	50
τ_2	20	80	80
τ_3	35	200	100
τ_4	62	90	81

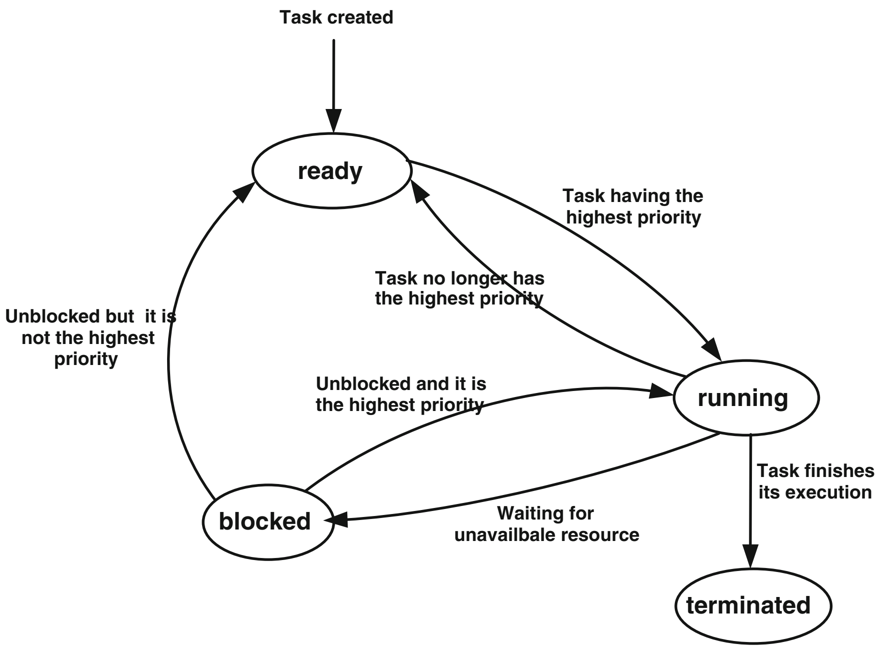
Task created
ready
Task having the highest priority
Task no longer has the highest priority
Unblocked but it is not the highest priority
Unblocked and it is the highest priority
running
Task finishes its execution
blocked
Waiting for unavailbale resource
terminated

Fig. 7 Task states

- a set of constraints ρ_i;
- a set of n_i Control Components ($n_i \geq 1$) such that the task τ_i is constituted by $CC_i^1, CC_i^2, \ldots, CC_i^{n_i}$.

One of the core components of an RTOS is the task scheduler which aims to determine which of the ready tasks should be executing. If there are no ready tasks at a given time, then no task can be executed, and the system remains idle until a task becomes ready (Fig. 8).

Running Example. *In the FESTO Benchmark Production System, when the task τ_1 is created, it is automatically marked as Ready task. At the instant $t1$,it is executed by the processor (i.e. it is in the Running state). When the task τ_1 needs a resource at the instant $t2$, it becomes blocked. Whenever the resource is available at the*

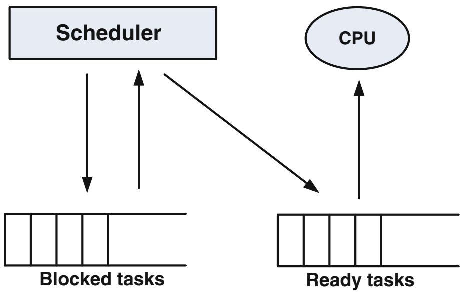

Fig. 8 Scheduling task

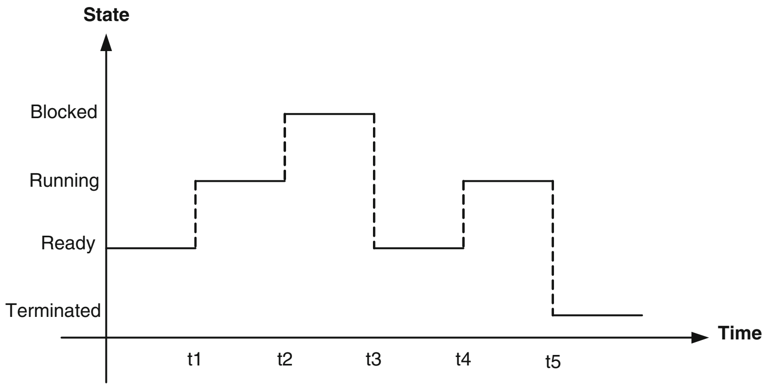

Fig. 9 The variation of states related to the task τ_1

instant $t3$, the task τ_1 is transformed into ready state. Finally, it is executed again since the time $t4$. It is terminated at the instant $t5$ (Fig. 9).

A scheduler related to a real-time operating system is characterized by (Fig. 10):

readyTask: a queue maintaining the set of tasks in ready state.
executingTask: a queue maintaining the set of tasks in executing state.
minPriority: the minimum priority assigned to a task.
maxPriority: the maximum priority assigned to a task.
timeSlice: the threshold of preempting a task (the quantity of time assigned to a task before its preemption).

Several tasks may be in the ready or blocked states. The system therefore maintains a queue of blocked tasks and another queue for ready tasks. The latter is maintained in a priority order, keeping the task with the highest priority at the top of the list.When a task that has been in the ready state is allocated the processor, it makes a state transition from ready state to running state. This assignment of the processor is called dispatching and it is executed by the dispatcher which is a part of the scheduler.

Running Example. *In the FESTO Benchmark Production System, we consider three tasks τ_1, τ_2 and τ_3. having as priority $p1, p2$ and $p3$ such that $p1 < p2 < p3$.*

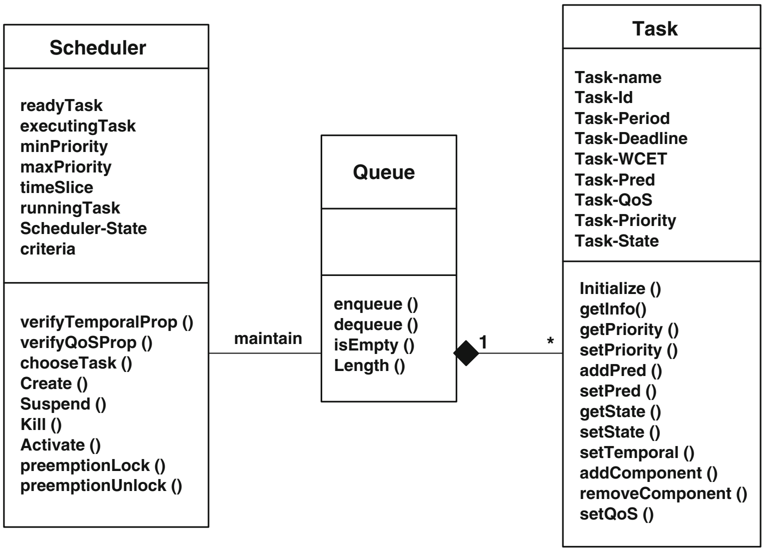

Fig. 10 The real time operating system

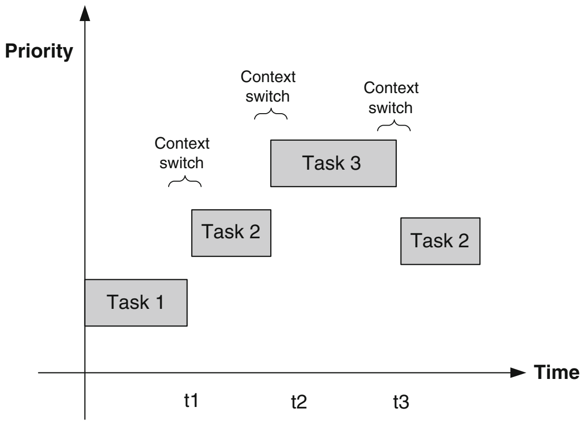

Fig. 11 The context switch between tasks

We suppose that the task τ_1 is running when the task τ_2 is created at the instant t1. As a consequence, there is a context switch so that the task τ_1 stays in a ready state and the other task τ_2 begins its execution as it has higher priority. At the instant t2, the task τ_3 which was already blocked waiting a resource, gets the resource. As the task τ_3 is the highest priority, the task τ_2 turns into ready state and τ_3 executes its routine. The task τ_3 continues processing until it has completed, the scheduler enables τ_2 to become running (Fig. 11).

5.3 Feasible and Safety Dynamic Reconfiguration of Tasks

We want to study the system's safety during reconfiguration scenarios. In fact, we want to keep tasks running while dynamically reconfiguring them. We assume for such system's task several software processes which provide functional services, and assume also reconfiguration processes that apply required modifications such as adapting connections, data or internal behaviors of the component. The execution of these different tasks is usually critical and can lead to incorrect behaviors of the whole system. In this case, we should schedule which process should be firstly activated to avoid any conflict between processes. Consequently, we propose in this section to synchronize processes for coherent dynamic reconfigurations applied to several tasks.

5.3.1 Reconfiguration and Service Processes

We want in this section to synchronize service and reconfiguration processes of a task according to the following constraints: (1) whenever a reconfiguration process is running, any new service process must wait until its termination; (2) a reconfiguration process must wait until the termination of all service processes before it begins its execution; (3) it is not possible to execute many reconfiguration processes in parallel; (4) several service processes can be executed at the same time. To do that, we use semaphores and also the famous synchronization algorithm between readers and writer processes such that executing a service plays the role of a reader process and reconfiguring a task plays the role of a writer process. In the following algorithm, we define *serv* and *reconfig* as semaphores to be initialized to 1. The shared variable *Nb* represents the number of current service processes associated to a specific task. Before the execution of a service related to a task, the service process increments the number *Nb* (which represents the number of service processes). It tests if it is the first process (i.e. *Nb* is equal to one). In this case, the operation P(reconfig) ensures that it is not possible to begin the execution if there is a reconfiguration process.

```
 P(serv)
 Nb ← NB + 1
  **if** (NB = 1) **then**
    P(reconfig)
 **end if**
V(serv)
```

After the execution of a service related to a task, the corresponding process decrements the number *Nb* and tests if there is no service process (i.e. *Nb* is equal to zero). In this case, the operation $V(reconfig)$ authorizes the execution of a reconfiguration process.

```
 P(serv)
 Nb ← NB − 1
```

```
    if (NB = 0) then
      V(reconfig)
    end if
V(serv)
```

Consequently, each service process related to a task does the following instructions:

Algorithm 2: execute a service related to a task

```
begin service
P(serv)
    Nb ← NB + 1
    if (NB = 1) then
        P(reconfig)
    end if
V(serv)

    execute the service

    P(serv)
    Nb ← NB - 1
    if (NB = 0) then
        V(reconfig)
    end if
V(serv)
end service
```

Running Example. *Let us take as a running example the task Test related to the EnAS system. To test a piece before elevating it, this component permits to launch the Test Service Process*. Figure 12 *displays the interaction between the objects Test Service Process,Service semaphore and Reconfiguration semaphore. The flow of events from the point of view of Test Service Process is the following:* (1) *the operation* $P(serv)$ *leads to enter in critical section for Service semaphore;* (2) *the number of services is incremented by one;* (3) *if it is the first service, then the operation* $P(reconfig)$ *permits to enter in critical section for Reconfiguration semaphore;* (4) *the operation* $V(serv)$ *leads to exit from critical section for Service semaphore;* (5) *the Test Service Process executes the corresponding service;* (6) *before modifying the number of service, the operation* $P(serv)$ *leads to enter in critical section for Service semaphore;* (7) *the number of services is decremented by one;* (8) *if there is no service processes, then the operation* $V(reconfig)$ *permits to exit from critical section for Reconfiguration semaphore;* (9) *the operation* $V(serv)$ *leads to liberate Service semaphore from its critical section.*

With the operation $P(reconfig)$, a reconfiguration process verifies that there is no reconfiguration processes nor service processes which are running at the same time. After that, the reconfiguration process executes the necessary steps and runs the operation $V(reconfig)$ in order to push other processes to begin their execution. Each reconfiguration process specific to a task realizes the following instructions:

Algorithm 3: reconfigure a task

```
begin reconfiguration
P(reconfig)
    execute the reconfiguration
V(reconfig)
end reconfiguration
```

Running Example. *Let us take as example the task Elevate related to EnAS system. The agent needs to reconfigure this task which permits to launch the Elevate Reconfiguration Process*. Figure 13 *displays the interaction between the following objects Elevate Reconfiguration Process and Reconfiguration semaphore. The flow of events from the point of view of Elevate Reconfiguration Process is the following:* (1) *the operation $P(reconfig)$ leads to enter in critical section for Reconfigura-*

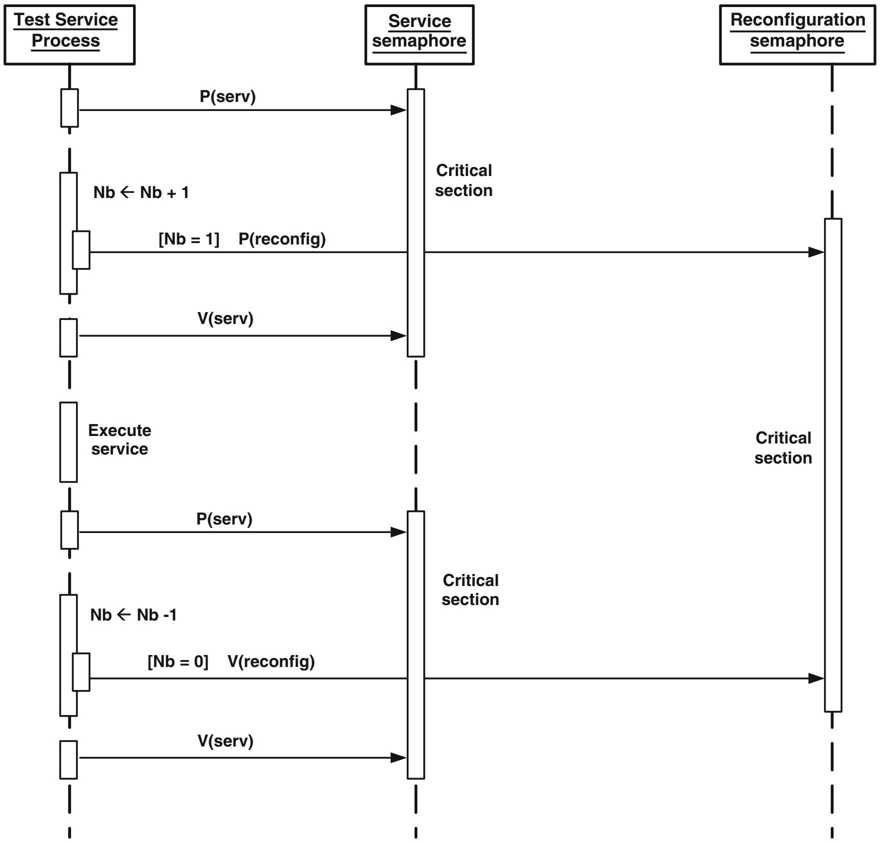

Fig. 12 The service process scenario

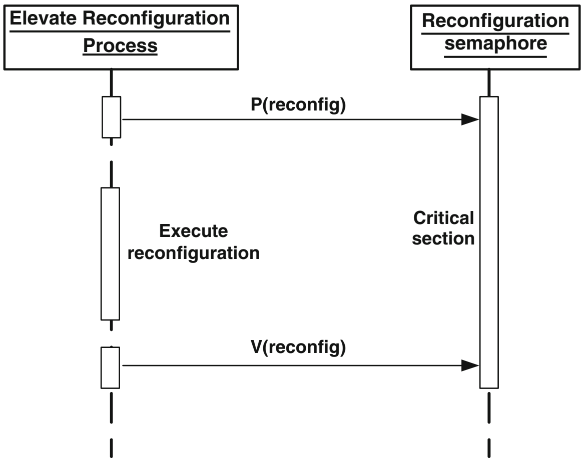

Fig. 13 The reconfiguration process scenario

tion semaphore; (2) *the Elevate Reconfiguration Process executes the corresponding reconfiguration;* (3) *the operation* $V(reconfig)$ *leads to liberate Reconfiguration semaphore from its critical section.*

5.3.2 Verification of Safety of the Synchronization

To verify the safety of the synchronization, we should verify if the different constraints mentioned above are respected.

First property: whenever a reconfiguration process is running, any service processes must wait until the termination of the reconfiguration. Let us suppose that there is a reconfiguration process (so the integer *reconfig* is equal to zero and the number of current services is zero). When a service related to this component is called, the number of current services is incremented (i.e. it is equal to 1) therefore the operation $P(reconfig)$ leads the process to be in a blocked state (as the integer *reconfig* is equal to zero). When the reconfiguration process terminates the reconfiguration, the operation $V(reconfig)$ permits to liberate the first process waiting in the semaphore queue. In conclusion, this property is validated.

Second property: whenever a service process is running, any reconfiguration processes must wait until the termination of the service. Let us suppose that there is a service process related to a component (so the number of services is greater or equal to one which means that the operation $P(reconfig)$ is executed and *reconfig* is equal to zero). When a reconfiguration is applied, the operation $P(reconfig)$ leads this process to be in a blocked state (as the *reconfig* is equal to zero). Whenever the number of service processes becomes equal to zero, the operation $V(reconfig)$ allows to liberate the first reconfiguration process waiting in the semaphore queue. As a conclusion, this property is verified.

Third property: whenever a reconfiguration process is running, it is not possible to apply a new reconfiguration process until the termination of the first one. Let us suppose that a reconfiguration process is running (so *reconfig* is equal to zero). Whenever, a new reconfiguration process tries to execute, the operation $P(\mathit{reconfig})$ puts it into a waiting state. After the reconfiguration process which is running is terminated, the operation $V(\mathit{reconfig})$ allows to liberate the first reconfiguration waiting process. Consequently, this property is respected.

Fourth property: whenever a service process is running, it is possible to apply another process service. Let us suppose that a service process $P1$ is running. Whenever, a new service process $P2$ tries to begin the execution, the state of $P2$ (activated or blocked) depends basically on the process $P1$:

- if $P1$ is testing the shared data *Nb*, then the operation $P(\mathit{serv})$ by the process $P2$ leads it to a blocking state. When the process $P1$ terminates the test of the shared data *Nb*, the operation $V(\mathit{serv})$ allows to launch the process waiting in the semaphore's queue.
- if $P1$ is executing its service, then the operation $P(\mathit{serv})$ by the process $P2$ allows to execute normally.

Thus, this property is validated.

5.4 Task Scheduling with Priority Ceiling Protocol

How to schedule periodic tasks with precedence and mutual exclusion constraints is considered as important as how to represent a task in a general real-time operating system. In our context, we choose the priority-driven preemptive scheduling used in the most real-time operating systems. The semaphore solution can lead to the problem of priority inversion which consists that a high priority task can be blocked by a lower priority task. To avoid such problem, we propose to apply the priority inheritance protocol proposed by Sha et al. [49].

The priority inheritance protocol can be used to schedule a set of periodic tasks having exclusive access to common resources protected by semaphores. To do so, each semaphore is assigned a priority ceiling which is equal to the highest priority task using this semaphore. A task τ_i is allowed to enter its critical section only if its assigned priority is higher than the priority ceilings of all semaphores currently locked by tasks other than τ_i.

Schedulability test for the priority ceiling protocol: a set of n periodic tasks using the priority ceiling protocol can be scheduled by the rate-monotonic algorithm if the following inequalities hold, $\forall i,\ 1 \le i \le n$,

$$C_1/T_1 + C_2/T_2 + \cdots + C_i/T_i + B_i/T_i \le i(2^{1/i} - 1)$$

where B_i denotes the worst-case blocking time of a task τ_i by lower priority tasks.

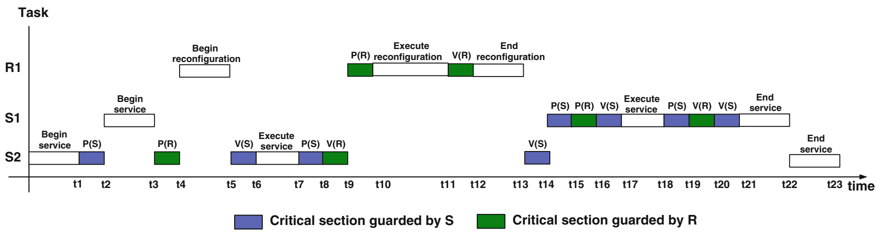

Fig. 14 The priority ceiling protocol applied to three tasks R1, S1 and S2

Table 3 The event and its corresponding action in Fig. 14

Event	Action
t0	S2 begins execution
t1	S2 locks S. The task S2 inherits the priority of S
t2	The task S1 is created. As it has more priority than S2, it begins its execution
t3	The task S1 fails to lock S as its priority is not higher than the priority ceiling of the locked S. The task S2 resumes the execution with the inherited priority of S
t4	The task S2 locks R. The task S2 inherits the priority of R. The task R1 is created and preempts the execution of S2 as it has the highest priority
t5	The task R1 fails to lock R as its priority is not higher than the priority ceiling of the locked R. The task S2 resumes the execution of the critical section
t6	The task S2 unlocks S
t7	The task S2 executes a service
t8	The task S2 locks S
t9	The task S2 unlocks R and therefore has as priority the same as S. The task R1 becomes having the highest priority. As it has more priority than S2, it resumes its execution
t10	The task R1 locks R
t11	The task R1 executes the reconfiguration
t12	The task R1 unlocks R
t13	The task R1 terminates its execution
t14	The task S2 unlocks S (thus S2 becomes having the lowest priority). Therefore, the task S1 resumes its execution
t15	The task S1 locks S
t16	The task S1 locks R
t17	The task S1 unlocks S
t18	The task S1 executes its service
t19	The task S1 locks S
t20	The task S1 unlocks R
t21	The task S1 unlocks S
t22	The task S1 achieves its execution
t23	The task S2 resumes the execution and terminates its service

Running Example. *In the FESTO Benchmark Production System, we consider three tasks $R1$ (a reconfiguration task), $S1$ and $S2$ (service tasks) having as priority*

$p1, p2$ and $p3$ such that $p1 > p2 > p3$. The sequence of processing steps for each task is as defined in the section previous paragraph where S (resp. R) denotes the service (resp. reconfiguration) semaphore:

$$R1 = \{\ldots P(R)\, execute\, reconfiguration\, V(R) \ldots\}$$
$$S1 = \{\ldots P(S) \ldots P(R) \ldots V(S)\, execute\, service\, P(S) \ldots V(R) \ldots V(S) \ldots\}$$
$$S2 = \{\ldots P(S) \ldots P(R) \ldots V(S)\, execute\, service\, P(S) \ldots V(R) \ldots V(S) \ldots\}$$

Therefore, the priority ceiling of the semaphore R is equal to the task R1 (because the semaphore R is used by the tasks R1, S1 and S2 and we know that the task R1 is the highest priority) and the priority ceiling of the semaphore S is equal to the task S1 (because the semaphore S is used by the tasks S1 and S2 and the priority task of S1 is higher). We suppose that the task $S2$ is running when the task $S1$ is created at the instant t3. We suppose also that the task $R1$ is created at the instant t5. Fig. 14*, a line in a high level indicates that the task is executing, a line in a low level indicates that the the task is blocked or preempted by another task.* Table 3 *explains more in details the example.*

6 Conclusion

This chapter deals with Safety Reconfigurable Embedded Control Systems. We propose conceptual models for the whole component-based architecture. We define a multi-agent architecture where a Reconfiguration Agent is affected to each device of the execution environment to handle local automatic reconfigurations, and a Coordination Agent is defined to guarantee safe distributed reconfigurations. To deploy a Control Component in a Real-Time Operating System, we define the concept of real-time task in general (especially its characteristics). The dynamic reconfiguration of tasks is ensured through a synchronization between service and reconfiguration processes to be applied. We propose to use the semaphore concept for this synchronization such that we consider service processes as readers and reconfiguration processes as writers. We propose to use the priority ceiling protocol as a method to ensure the scheduling between periodic tasks with precedence and mutual exclusion constraints. The main contributions presented through this work are: the study of Safety Reconfigurable Embedded Control Systems from the functional to the operational level and the definition of a real-time task independently from any real-time operating system as well as the scheduling of these real-time tasks considered as periodic tasks with precedence and mutual exclusion constraints. The chapter's contribution is applied to two Benchmark Production Systems FESTO and EnAS available at Martin Luther University in Germany.

References

1. M. Akerholm, J. Fredriksson, *A Sample of Component Technologies for Embedded Systems*
2. Y. Al-Safi, V. Vyatkin, *An ontology-based reconfiguration agent for intelligent mechatronic systems* (Springer, New York, 2007). Third International Conference on Industrial Applications of Holonic and Multi-Agent Systems
3. C. Angelov, K. Sierszecki, N. Marian, Design models for reusable and reconfigurable state machines, in *EUC 2005, LNCS 3824* eds. by L.T. Yang et al., International Federation for Information Processing (2005), pp. 152–163
4. C. Bidan, V. Issarny, T. Saridakis, A. Zarras, A dynamic reconfiguration service for CORBA, in *CDS 98: Proceedings of the International Conference on Configurable Distributed Systems* (IEEE Computer Society, 1998)
5. K.-J. Cho, et al., A study on the classified model and the agent collaboration model for network configuration fault management. Knowl. Based Syst., 177–190 (2003)
6. M. Colnaric, D. Verber, W.A. Halang, A data-centric approach to composing embedded, real-time software components. J. Syst. Softw. **74**, 25–34 (2005)
7. F. Cottet, J. Delacroix, C. Kaiser, Z. Mammeri, *Scheduling in Real-Time Systems* (Wiley, New York, 2002)
8. I. Crnkovic, *Component-based Approach for Embedded Systems* (2003)
9. I. Crnkovic, M. Larsson, *Building Reliable Component-based Software Systems* (Artech House, 2002)
10. I. Crnkovic, M. Larsson, *Grid Information Services for Distributed Resource Sharing* (Artech House, UK, 2002). Building reliable component-based software systems
11. M. de Jonge, Developing Product Lines with Third-Party Components. Electronic Notes in Theoretical Computer Science (2009), pp. 63–80
12. EN50126, *Railway Applications the Specification and Demonstration of Dependability, Reliability, Availability, Maintainability and Safety (RAMS)* (Comite Europeen de Nomalisation Electrotechnique, 1999)
13. EN50128, *Railway Applications Software for Railway Control and Protection Systems* (Comite Europeen de Nomalisation Electrotechnique, 2002)
14. EN50129, *Railway Applications Safety Related Electronic Systems for Signalling* (Comite Europeen de Nomalisation Electrotechnique, 2002)
15. EN954, *Safety of Machinery Safety-related Parts of Control Systems* (Comite Europeen de Nomalisation Electrotechnique, 1996)
16. R. Faller, Project experience with IEC 61508 and its consequences. Saf. Sci. **42**, 405–422 (2004)
17. T. Genler, O. Nierstrasz, B. Schonhage, *Components for Embedded Software The PECOS Approach*
18. T. Genssler, et al., *PECOS in a Nutshell* (2002)
19. A. Gharbi, H. Gharsellaoui, M. Khalgui, S. Ben Ahmed, Functional safety of distributed embedded control systems, in *Handbook of Research on Industrial Informatics and Manufacturing Intelligence: Innovations and Solutions*, eds. by M.A. Khan, A.Q. Ansari (2011)
20. A. Gharbi, M. Khalgui, S. Ben Ahmed, Functional safety of discrete event systems. First Workshop of Discrete Event Systems (2011)
21. A. Gharbi, M. Khalgui, S. Ben Ahmed, Inter-agents communication protocol for distributed reconfigurable control software components. The International Conference on Ambient Systems Networks and Technologies (ANT), 8–10 Nov 2010
22. A. Gharbi, M. Khalgui, S. Ben Ahmed, Model checking optimization of safe control embedded components with refinement. 5th International conference on Design and Technology of Integrated Systems in Nanoscale Era (2010)
23. A. Gharbi, M. Khalgui, S. Ben Ahmed, Optimal model checking of safe control embedded software components. 15th IEEE International Conference on Emerging Technologies and Factory Automation (2010)

24. A. Gharbi, M. Khalgui, H.M. Hanisch, Functional safety of component-based embedded control systems. 2nd IFAC Workshop on Dependable Control of Discrete Systems (2009)
25. http://www.program-Transformation.org/Tools/KoalaCompiler. Last accessed on 11 July 2010
26. IEC 1131–3, *Programmable Controllers, Part 3: Programming Languages* (International Electrotechnical Commission, Geneva, 1992)
27. IEC 61508, *Functional Safety of Electrical/Electronic Programmable Electronic Systems: Generic Aspects*. Part 1: General requirements (International Electrotechnical Commission, Geneva, 1992)
28. IEC60880, *Software for Computers in the Safety Systems of Nuclear Power Stations* (International Electrotechnical Commission, 1987)
29. IEC61511, *Functional Safety: Safety Instrumented Systems for the Process Industry Sector* (International Electrotechnical Commission, Geneva, 2003)
30. IEC61513, *Nuclear Power Plants Instrumentation and Control for Systems Important to Safety General Requirements for Systems* (International Electrotechnical Commission, Geneva, 2002)
31. G. Jiroveanu, R.K. Boel, A distributed approach for fault detection and diagnosis based on Time Petri Nets. Math. Comput. Simul., 287–313 (2006)
32. M. Kalech, M. Linder, G.A. Kaminka, Matrix-based representation for coordination fault detection: a formal approach. Comput. Vis. Image Underst.
33. A. Ketfi, N. Belkhatir, P.Y. Cunin, *Automatic Adaptation of Component-based Software Issues and Experiences* (2002)
34. M. Khalgui, H.M. Hanisch, A. Gharbi, Model-checking for the functional safety of control component-based heterogeneous embedded systems. 14th IEEE International conference on Emerging Technology and Factory Automation (2009)
35. J. Kramer, J. Magee, The evolving Philosophers problem: dynamic change management. IEEE Trans. Softw. Eng. **16** (1990)
36. P. Leitao, Agent-based distributed manufacturing control: A state-of-the-art survey. Eng. Appl. Artif. Intell. (2008)
37. A.J. Massa, *Embedded Software Development with eCos*, 1st edn (Prentice Hall, Upper Saddle River, NJ, USA, 2002)
38. S. Merchant, K. Dedhia, *Performance Comparison of RTOS* (2001)
39. C. Muench, *The Windows CE Technology Tutorial: Windows Powered Solutions for the Developer* (Addison Wesley, Reading, 2000)
40. S. Olsen, J. Wang, A. Ramirez-Serrano, R.W. Brennan, Contingencies-based reconfiguration of distributed factory automation. Robot. Comput. Integr. Manuf., 379–390 (2005) (Safety Reconfigurable Embedded Control Systems 31)
41. G.A. Papadopoulos, F. Arbab, *Configuration and Dynamic Reconfiguration of Components Using the Coordination Paradigm* (2000)
42. P. Pedreiras, L. Almeida, *Task Management for Soft Real-Time Applications based on General Purpose Operating, System* (2007)
43. G. Pratl, D. Dietrich, G. Hancke, W. Penzhorn, A new model for autonomous, networked control systems. IEEE Trans. Ind. Inform. **3**(1) (2007)
44. QNX Neutrino, *Real Time Operating System User Manual Guide* (2007)
45. A. Rasche, A. Polze, *ReDAC—Dynamic Reconfiguration of distributed component-based applications with cyclic dependencies* (2008)
46. A. Rasche, W. Schult, Dynamic updates of graphical components in the .NET Framework, in *Proceedings of SAKS07 Workshop* eds. by A. Gharbi, M. Khalgui, M.A. Khan, vol. 30 (2007)
47. M.N. Rooker, C. Sunder, T. Strasser, A. Zoitl, O. Hummer, G. Ebenhofer, *Zero Downtime Reconfiguration of Distributed Automation Systems : The eCEDAC Approach* (Springer, New York, 2007). Third International Conference on Industrial Applications of Holonic and Multi-Agent Systems
48. G. Satheesh Kumar, T. Nagarajan, Experimental investigations on the contour generation of a reconfigurable Stewart platform. IJIMR **1**(4), 87–99 (2011)
49. L. Sha, R. Rajkumar, J.P. Lehoczky, Priority inheritence protocols: an approach to real-time synchronization. IEEE Trans. Comput. **39**(9), 1175–1185 (1990)

50. D.D. Souza, A.C. Wills, *Objects, Components and Frameworks: The Catalysis Approach* (Addison-Wesley, Reading, MA, 1998)
51. D.B. Stewart, R.A. Volpe, P.K. Khosla, Design of dynamically reconfigurable real-time software using port-based objects. IEEE Trans. Softw. Eng. **23**, 592–600 (1997)
52. C. Szyperski, D. Gruntz, S. Murer, *Component Software Beyond Object- Oriented Programming* (The Addison-Wesley Component Software Series, 2002)
53. R. van Ommering, F. van der Linden, J. Kramer, J. Magee, *The Koala Component Model for Consumer Electronics Software* (IEEE Computer, Germany, 2000), pp. 78–85
54. M. Winter, *Components for Embedded Software—The PECOS Approach*
55. R. Wuyts, S. Ducasse, O. Nierstrasz, A data-centric approach to composing embedded, real-time software components. J. Syst. Softw. (74), 25–34 (2005)