Safety Implementation of Adaptive Embedded Control Components

Atef Gharbi
National Institute of Applied Sciences and Technology, Tunisia
www.insat.rnu.tn
atef.elgharbi@gmail.com

Mohamed Khalgui
Xidian University, China
khalgui.mohamed@gmail.com

Samir Ben Ahmed
National Institute of Applied Sciences and Technology, Tunisia
www.insat.rnu.tn
samir.benahmed@fst.rnu.tn

The paper deals with dynamic reconfigurations of component-based adaptive embedded control systems to be automatically handled at run-time by intelligent agents. We define a Control Component as a software unit supporting control tasks of the system which is assumed to be a network of components with precedence constraints. We assume a reconfiguration scenario as any run-time operation allowing the addition, removal or update of software components to adapt the system to its environment. Several complex networks can implement the system such that each one is executed at a given time when a corresponding reconfiguration scenario is automatically applied by the agent. The latter is specified in our research by nested state machines to cover all reconfiguration forms of the software architecture, structure or data of the system. We propose technical solutions to implement the whole agent-based architecture, by defining UML meta-models for both Control Components and also agents. To guarantee safety dynamic reconfigurations at run-time, we define service and reconfiguration processes for components and use the semaphore concept to ensure safety mutual exclusions.

Adaptive Embedded Control System, Dynamic Reconfiguration, Software Control Component, Intelligent Agent, Semaphore.

1. INTRODUCTION

Nowadays in industry, different component-based technologies such as IEC61499 Function Blocks, PBO, Rubus, Koala, and so on Crnkovic (2002), and also Architecture Description Languages (abbr. ADL) Dissaux (2006) have been proposed to reuse already developed components and to follow modular approaches for specifications and validations of embedded control systems (ECS). To be independent of any technology or language, we define in Gharbi (2009) a Control Component as a software unit owning data to support system's functionalities. The component is composed of an implementation and interfaces for external interactions. An embedded control system is assumed to be composed of interactive components sharing controls of physical processes. The new generations of ECS address new criteria as flexibility and agility. To reduce their development costs, these systems should be reconfigured at run-time to adapt their behaviors to their environment. We define in Gharbi (2009) an agent-based architecture for these systems such that an intelligent agent is assumed to dynamically handle reconfiguration scenarios. The agent allows to add, remove or update components to adapt the system. It is composed of four units: (1) Architecture Unit that changes the system's architecture by changing the current subset of Control Components, (2) Control Unit that updates compositions of components for a particular software architecture, (3) Implementation Unit that updates at run-time the implementation of components, (4) Data Unit that applies light reconfigurations of data in the system. For more details about this agent's model, we refer to Khalgui (2009).

The subject of this paper is to extend all our previous papers by proposing a specific methodology to handle dynamic reconfigurations in adaptive embedded control system using intelligent agents. We propose in particular 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. We define service processes as software processes for components to provide system's functionalities, and define reconfiguration processes as agent's 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 readers 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 note that all the paper contributions are applied to a Benchmark Production System available at Martin Luther University in Germany.

In the next Section, we introduce the benchmark production system (EnAS) to be followed in the paper as a running example. Section 3 presents a background (introducing our previous work) and an analysis of related works dealing with dynamic reconfigurations. We study the software Control Components in Section 4 (we present in particular the meta-model for a Control Component as well as the conceptual model for Component-based Embedded Control Systems). In Section 5, we propose the software architecture of intelligent agents. In Section 6, we study feasible and safety dynamic reconfigurations through semaphore-based synchronization of processes. Finally, we conclude the paper in Section 7.

2. BENCHMARK PRODUCTION SYSTEM

We consider as a running example in this research work the benchmark production system EnAS (Website: http://at.iw.uni-halle.de/forschung/enas_demo) available in the research laboratory of Prof. Hanisch at Martin Luther University in Germany. This system transports pieces from production systems to storing units. The pieces shall be placed inside tins to be closed with caps. Two different production strategies are assumed to be applied in this paper: 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 the storing units (Figure 1).

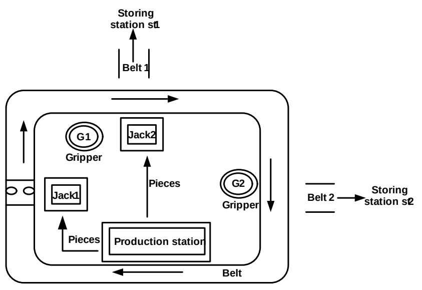

Figure 1: *The benchmark production system*

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.

Running Example. *The EnAS Benchmark Production System is composed of 7 control components:* $Move\_Belt$, Put_Piece, $Move\_Belt1$, $Test$, $Detect$, $Elevate$ *and* $Tester\_Failure$ *(Figure 2).*

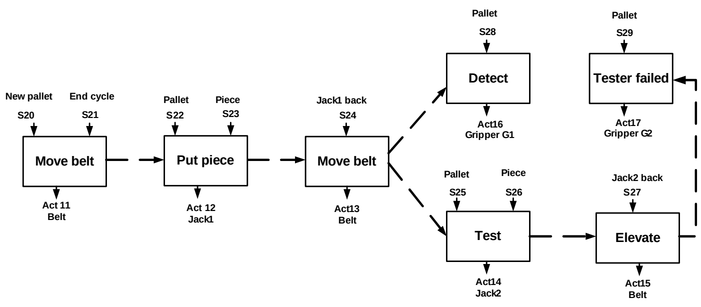

Figure 2: *The control components related to EnAS system*

3. BACKGROUND & RELATED WORK

In this section, we describe our previous contributions and related works to prove the originality of the current paper.

3.1. Background

Until now, we have published a package of previous papers dealing with reconfigurable embedded control systems. We define in Gharbi (2009) the concept of Control Components as generic independent components of any component-based technology. The use of intelligent agent in our research work is well justified for the following reasons: (i) the control of physical plant through collecting data, selecting the appropriate action and commanding the actuators; (ii) the collective behavior enabling the interaction with other entities through a communication protocol; (iii) the fault treatment as well as the reconfiguration of the system whenever it is necessary to do so must be effectuated with intelligence.
We define in Khalgui (2009) a new software architecture for intelligent agents to control and adapt systems to their environments. Each agent is composed of four levels: Architecture Unit, Control Unit, Implementation Unit and Data Unit . To verify the correctness of its behavior, we specify the whole architecture according to the formalism Net Condition/Event Systems which is an extension of Petri Nets. To avoid the combinatory explosion, we apply in Gharbi (2010a), Gharbi (2010b) a refinement-based approach to verify step by step subsets of components. To guarantee correct and feasible distributed reconfigurations, we define in Gharbi (2010c), Gharbi (2011b) an inter-agents communication protocol. We study in Gharbi (2011a) the fault management by intelligent agents. In this article, we continue our research by proposing an original implementation of this agent-based architecture.

3.2. Agent's Structure

We define in Khalgui (2009) the following units that belong to four hierarchical levels of the proposed intelligent agent's architecture: (i) **First level:** (**Architecture Unit**) this unit checks the plant evolution and changes the system's software architecture (adds/removes Control Components) when particular conditions are satisfied, (ii) **Second level:** (**Control Unit**) for a particular *loaded* software architecture, this unit checks the plant's evolution and reconfigures compositions of corresponding Control Components, (iii) **Third level:** (**Implementation Unit**) for a particular composition of Control Components, this unit reconfigures their implementations, (iv) **Fourth level:** (**Data Unit**) this unit updates data if particular conditions are satisfied.
The design of complex behavior with state machine is a hard task as there is a multitude of states. To avoid this problem, we propose the use of nested state machine. We design the agent in Gharbi (2009) by nested state machines where the Architecture Unit is specified by an Architecture State Machine (denoted by ASM) in which each state corresponds to a particular software architecture. Therefore, each transition of the ASM corresponds to the load (or unload) of Control Components into (or from) the memory. We construct for each state $S1$ of the ASM a particular Control State Machine (denoted by CSM) in the Control Unit. This state machine specifies all compositions of Control Components when the system software architecture corresponding to the state $S1$ is loaded. Each transition of any CSM has to be fired if particular conditions are satisfied. For each state $S2$ of a state machine CSM, we define in the Implementation Unit a particular Implementation State Machine (denoted by ISM) in which each state defines particular implementations of Control Components that we compose in the state $S2$. Finally, the Data Unit is specified also by Data State Machines (denoted by DSMs) that define all possible values of data in the system. The main topic of the current paper is to extend our previous works by studying the implementation of this agent-based architecture.

Running Example. *We design the agent in the EnAS system by nested state machines as depicted in Figure 3. The first level is specified by ASM where each state defines a particular software architecture of the system (i.e. a particular composition of blocks to be loaded in the memory). The state S_1 (resp. S_2) corresponds to the Second (resp. First) Production Policy where the components CC_J_1, CC_J_2 and CC_G_2 (resp. only CC_J_1 and CC_G_1) are loaded in the memory. We associate for each one of these states a CSM state machine in the Control Unit. We assume in addition for the states S_{12} and S_{13} two ISM state machines in the implementation Unit. Finally, the Data Unit is specified by a DSM state machine defining the values that $Threshold$ (denoted by p in the figure 3) takes under well-defined conditions. Note that if we follow the Second Production Policy (state S_1) and the gripper G_2 is broken, then we have to change the policy and also the system's architecture by loading the Control Component CC_G_1 to remove pieces into $Belt1$. On the other hand, we associate in the second level for the state S_1 the CSM S_1 defining the different reconfiguration forms to apply when the first software architecture is loaded in the memory. In particular,* ***if*** *the state S_{11} is active and the Jack station J_1 is broken,* ***then*** *we activate the state S_{12} in which the Jack station J_2 is running alone to place only one piece in the* *tin. In this case, the internal behavior of the Control Component CC_Belt has to be changed (i.e. the tin has to be transported directly to the station J_2). In the same way, **if** we follow the same policy in the state S_{11} and the Jack station J_2 is broken, **then** we have to activate the state S_{13} where the system uses the Jack station J_1 and the Gripper station G_2 to put pieces in tins. In this case, two implementations are possible: (1) **If** $nb_{pieces}/nb_{tins+caps} > Threshold$ ($nb_{pieces}/nb_{tins+caps}$ is denoted by q in the figure 3), **then** the Jack station J_1 puts a first piece in the tin that has to follow a complete round in the EnAS platform to charge in the same station a second piece (state S_{131}). (2) **Otherwise** the Jack station J_1 puts a unique piece in the tin to be removed thereafter by G_2 (state S_{132}). The system implementation in the states S_{121} and S_{122} (or in S_{131} and S_{132}) is different. Finally, we specify in the Data Unit a DSM state machine where we change the value of $Threshold$ when the Gripper station G_1 or G_2 is broken (we suppose as an example that we are not interested in the system performance when the Gripper G_1 or G_2 is broken).*

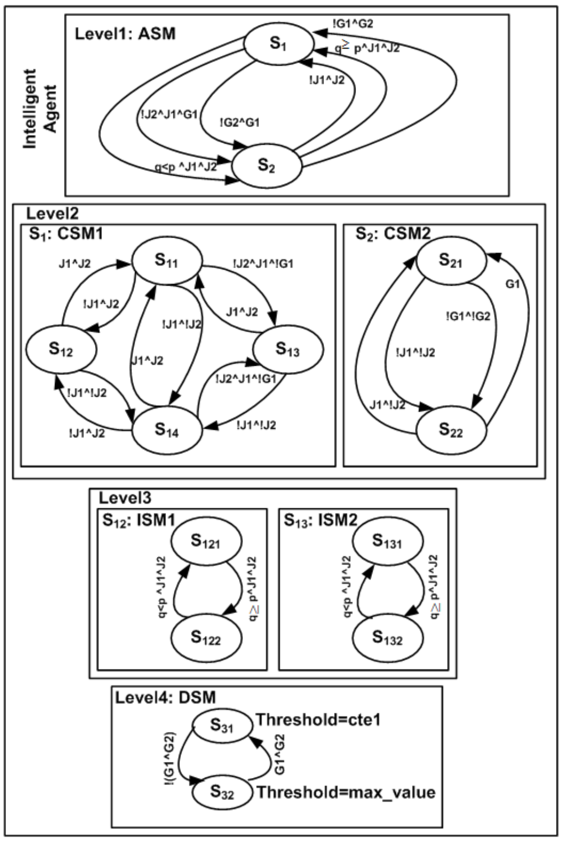

Figure 3: *Agent specification with nested state machines*

3.3. Related works

The new generation of industrial control systems is addressing today new criteria as flexibility and agility Pratl (2007); Rezg (2001). 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 Marian (2005), whereas dynamic reconfigurations are dynamically applied at run-time. Two cases exist in the last policy: manual reconfigurations applied by users strasser (2007) and automatic reconfigurations applied by intelligent agents Vyatkin (2007). 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. Kramer (1990) 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. Bidan (1998) treat problems of dynamic reconfigurations in CORBA. They 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. Papadopoulos (2001) propose a dynamic reconfiguration language based on features. They 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. Rasche (2007) propose to apply dynamic updates on graphical components (for example button, graphical interface, ...) in a .Net framework. To do so, they 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. Rasche (2008) define in addition 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 paper is the study of dynamic reconfiguration with semaphore ensuring the following points: (i) blocking connections without blocking involved components; (ii) safety and correctness of the proposed solution; (iii) independence of any specific language; (iv) verification of consistency (i.e. logical constraints) delegated to the software agent; (v) suitable for large-scale applications.

4. SOFTWARE CONTROL COMPONENTS

To be independent of any language or industrial technology, we propose the concept of Control Components in Gharbi (2009) as software units of adaptive embedded control systems. A component is classically defined as "a coherent package of software that can be independently developed and delivered, and that offers interfaces to be connected, unchanged, with other components in order to compose a larger system" Souza (1998). In the following paragraphs, we extend our previous published papers by proposing conceptual meta-models for Control Components.

4.1. Conceptual Meta-Model for Control Components

An embedded control system is assumed to be a set of control components with precedence constraints, to allow controls of devices by reading and interpreting data from sensors, before reacting and activating corresponding actuators. Three concepts have to be defined to compose a system: Control Components, interfaces and connectors. A "Control Component" is defined as an event-triggered software unit owning data to execute system's functionalities. An interface has a main role to define "components access points" SZYPERSKI (2002). By believing that internal designs and implementations of components are hidden, the corresponding interfaces allow their external interactions. An interface specifies all signatures of operations to be provided or required by a component. Therefore, remote components have accesses through its interfaces. A Control Component may have different interfaces such that each one defines offered services. Furthermore, the interface of a component is independent of the corresponding implementation. Consequently, it is easy to modify the latter without changing the interface and vice-versa. The connector is used to ensure connections between provided and required interfaces of components (for example data transmission). A principal characteristic of connectors is to link interfaces with complementary roles. In the current paper, we are interested in the original design and implementation of components. As a new contribution in the current paper, we propose the conceptual meta-model of Control Components in Figure 4 where the classes *Control Component*, *Interface* and *Connector* are distinguished. Note that an object *Control Component* may be composed of several sub-components. Moreover, it may have one or many *Interface* objects. Note also that an *Interface* object may be a *Provided* or *Required* interface. A *Provided* interface presents services for external components, whereas a *Required* interface defines the need of a service from remote components. An object of type *Connector* is a link between *Provided* and *Required* objects. A connector is especially deployed when it links distributed components on different devices. Finally, the classes *Control Component* and *Interface* provide some operations which may be represented by *Operation* class in general (or more specific with *Component Operation* and *Interface Operation* classes). *Component Operations* are for example creating or destroying Control Components, whereas *Interface Operations* are connecting and disconnecting between two interfaces.

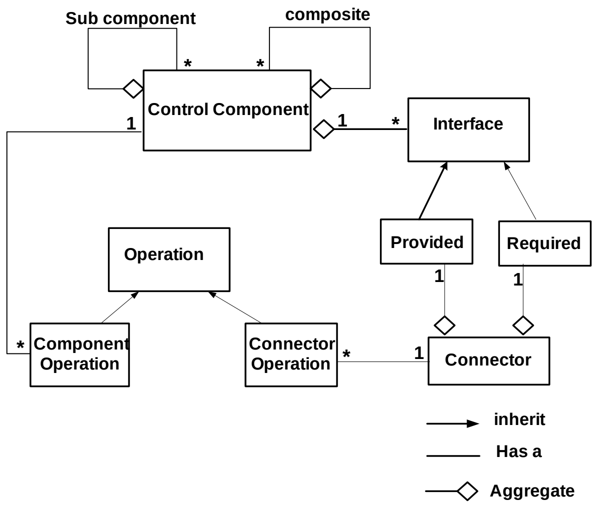

Figure 4: *The composition Meta-model of a Control Component*

Running Example. *We show in Figure 5 a static view describing possible instances that are created from different classes. The $PutPiece$ object, which is an instance of $Control\ Component$ class, has as operation $PutOp$ object (instance of $Component\ Operation$ class). The $PutDone$ object, which is an instance of $Interface\ Provided$ class, represents data to be sent to the required interface of succ($PutPiece$), for which we associate the connector object $c2$. The $MoveDone$ object, which is an instance of $Interface\ Required$ class, represents data to be received from the provided* *interface of pred($PutPiece$), for which we associate the connector object $c1$.*

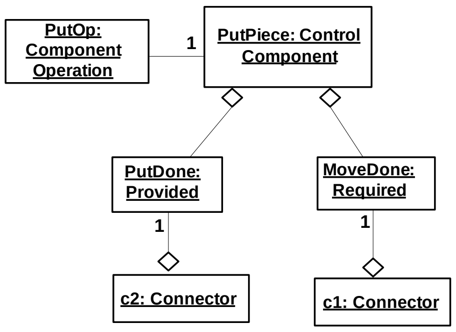

Figure 5: *The object diagram of $PutPiece$ Control Component*

4.2. Conceptual Model of Component-based Embedded Control Systems

We define in this paper a new implementation of component-based embedded control systems. We propose according to the well-known UML language the concepts of sensors, actuators, Control Components, Execution Traces, Execution Trace Networks and Systems (Figure 6).

- **Sensor:** is defined by the following information: (i) Name: the identifier of the sensor, (ii) Min: the minimum value to be provided by the sensor, and (iii) Max: the maximum value to be provided by the sensor.
- **Actuator:** each actuator is characterized by the following data: (i) Name: the identifier of the actuator, and (ii) Period: the execution period,
- **Control Component:** is represented as follows: (i) Name: the name of the control component, (ii) Min: the minimum value accepted by the control component, (iii) Max: the maximum value accepted by the control component, (iv) List of sensors: the list of sensors related to the control component, and (v) Actuator: the actuator associated to the control component,
- **Execution Trace:** represents the solution to be applied. The Execution Trace is represented as follows: (i) Name: the name of the execution trace, and (ii) listCC: the list of Control Components,
- **Execution Trace Network:** represents different reconfigurations that can be applied for a given policy. The Execution Trace Network is represented as follows: (i) Name: the name of the execution trace network, and (ii) listET: the list of execution traces,
- **System:** is represented by different policies that can be applied. The System is represented as follows: (i) Name: the name of the execution trace network, and (ii) listNetwork: the list of execution trace networks.

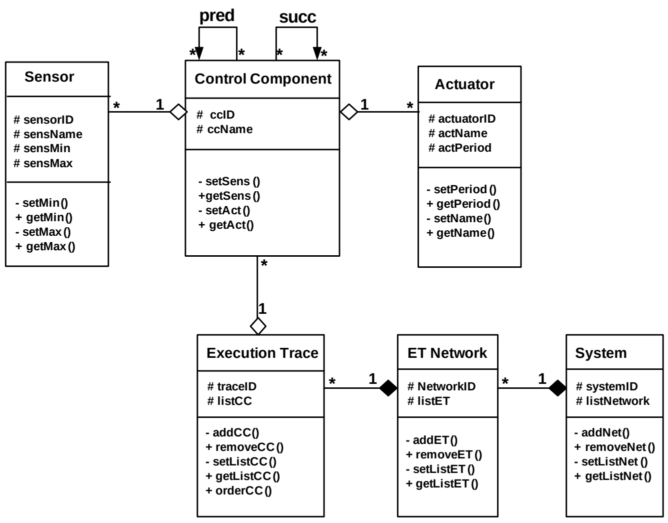

Figure 6: *The general conception of an application based on control components*

Running Example. *We present in Figure 7 a static view where we show the $EnAS$ object which is an instance of $System$ class, and which is constituted by only one $Network$ object (instance of $ETNetwork$ class). The Execution Trace Network is composed of two execution traces: The first execution trace ($ET1$) is composed of the following objects: $MoveBelt$, $PutPiece$, $MoveBelt1$, and $Detect$. The second execution trace ($ET2$) contains the following objects: $MoveBelt$, $PutPiece$, $MoveBelt1$, $Test$, $Elevate$ and $TesterFailure$.*

5. AGENT-BASED DYNAMIC RECONFIGURATIONS

We are interested in safety component-based embedded control systems. To guarantee their behavioral safety, we propose in Khalgui (2009) an agent-based architecture in which an intelligent agent is defined to check and adapt the system at run-time by applying dynamic reconfiguration scenarios. In this paper, we extend this research by proposing a new design and implementation of this architecture.

5.1. 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

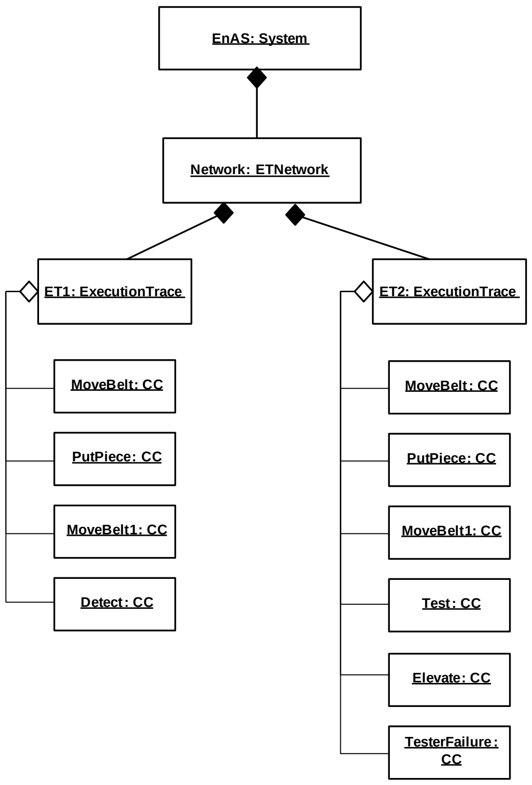

Figure 7: *The Object Diagram of EnAS System*

dynamically controls the plant, we use nested state machines in which states correspond to state machines. We propose a conceptual model for a nested state machine in Figure 8 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.

The key elements to be presented are :

- **Nested State Machine:** is constituted by a set of State machines ($listSM$). It has an initial state machine ($initialSM$) and it is characterized by a set of Input/Output events ($inputEvent$ and $outputEvent$).
- **State machine:** composed of different states ($listStates$). It has its own initial state ($initialState$) as well as its Input/Output events ($inputEvent$ and $outputEvent$).

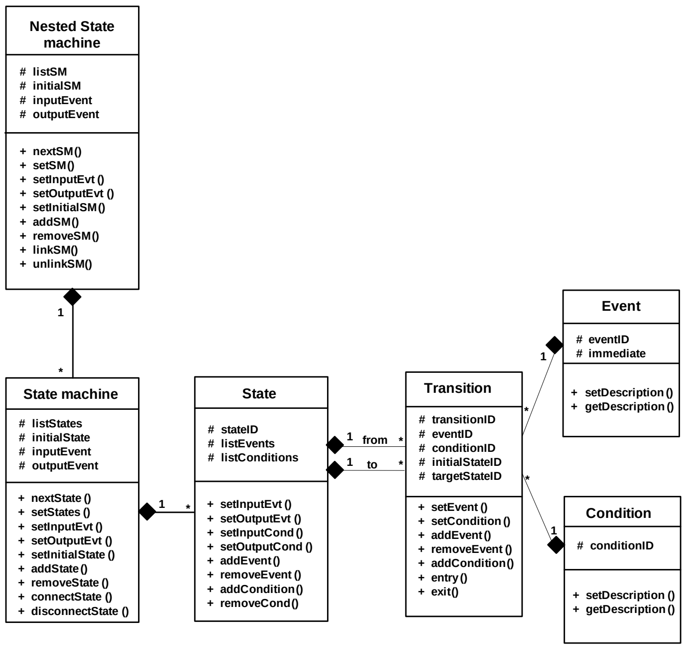

Figure 8: *The Meta-model nested state machine*

- **State:** is identified by a $stateID$. For each state, there are some events ($listEvents$) and conditions ($listEvents$) associated to it.
- **Transition:** is identified by a $transitionID$. A transition is ensured if only some specific condition ($conditionID$) and event ($eventID$) take place. A transition corresponds to a link between a state ($initialStateID$) to another ($targetStateID$).

Running Example. *On the one hand, we distinguish in Figure 9 the objects ASM, $CSM1$, $CSM2$, $ISM1$, $ISM2$, and DSM instantiated from State Machine class. On the other hand, we distinguish also the objects $System$, CSM and ISM instantiated from Nested State Machine class. The $System$ object contains the ASM, CSM, ISM and DSM objects. The CSM object contains in its turn the $CSM1$ and $CSM2$ objects. The object ISM contains in its turn the $ISM1$ and $ISM2$ objects.*

5.2. Conceptual Architecture For Intelligent Agents

We propose a generic architecture for intelligent agents depicted in Figure 10. This architecture consists of the following parts: (i) the Event Queue to save different input events that may take place in the system, (ii) the intelligent software agent that reads an input event from the Event Queue and reacts as soon as possible, (iii) the set of state machines such that each one is composed of a set of states, (iv) each state represents a specific information about the system. The agent,

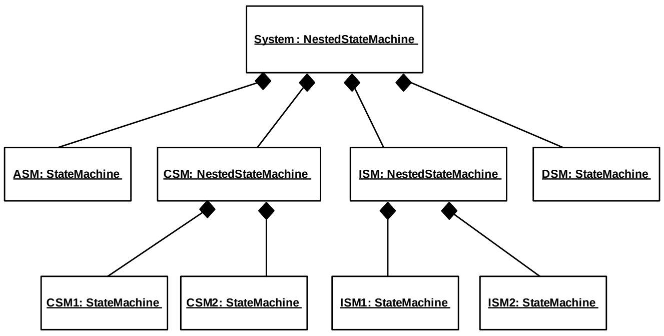

Figure 9: *Object Diagram related to EnAS Nested State Machine*

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: (i) 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. (ii) 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. (iii) The agent is not supposed to know components that it has to add or remove in a reconfiguration case.

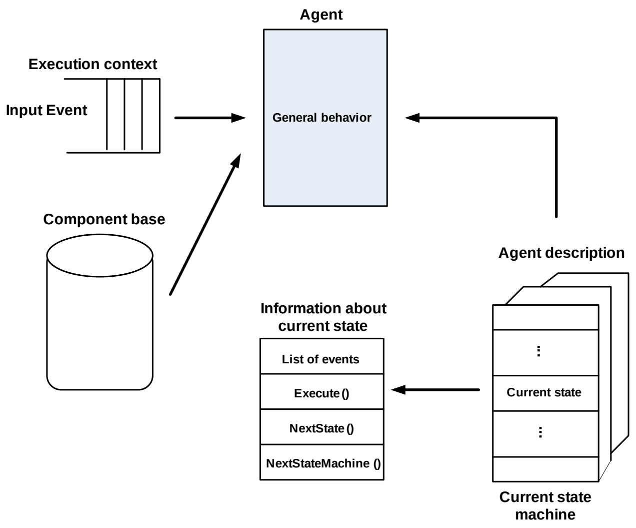

Figure 10: *The internal agent behavior*

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.
```

6. FEASIBLE AND SAFETY DYNAMIC RECONFIGURATIONS

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

6.1. Reconfiguration & Service Processes

We want in this section to synchronize service and reconfiguration processes of a component according to the following constraints: (i) whenever a reconfiguration process is running, any new service process must wait until its termination; (ii) a reconfiguration process must wait until the termination of all service processes before it begins its execution; (iii) it is not possible to execute many reconfiguration processes in parallel; (iv) 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 component 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 Control Component. Before the execution of a service related to a component, 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 Control Component, 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 Control Component does the following instructions:

Algorithm 2: execute a service related to a Control Component

```
begin
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.
```

Running Example. *Let us take as a running example the Control Component $Test$ related to the EnAS system. To test a piece before elevating it, this component permits to launch the Test Service Process. Figure 11 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: (i) the operation $P(serv)$ leads to enter in critical section for Service semaphore; (ii) the number of services is incremented by one; (iii) if it is the first service, then the operation $P(reconfig)$ permits to enter in critical section for Reconfiguration semaphore; (iv) the operation $V(serv)$ leads to exit from critical section for Service semaphore; (v) the Test Service Process executes the corresponding service; (vi) before modifying the number of service, the operation $P(serv)$ leads to enter in critical section for Service semaphore; (vii) the number of services is decremented by one; (viii) if there is no service processes, then the operation $V(reconfig)$ permits to exit from critical section for Reconfiguration semaphore; (ix) the operation $V(serv)$ leads to liberate Service semaphore from its critical section.*

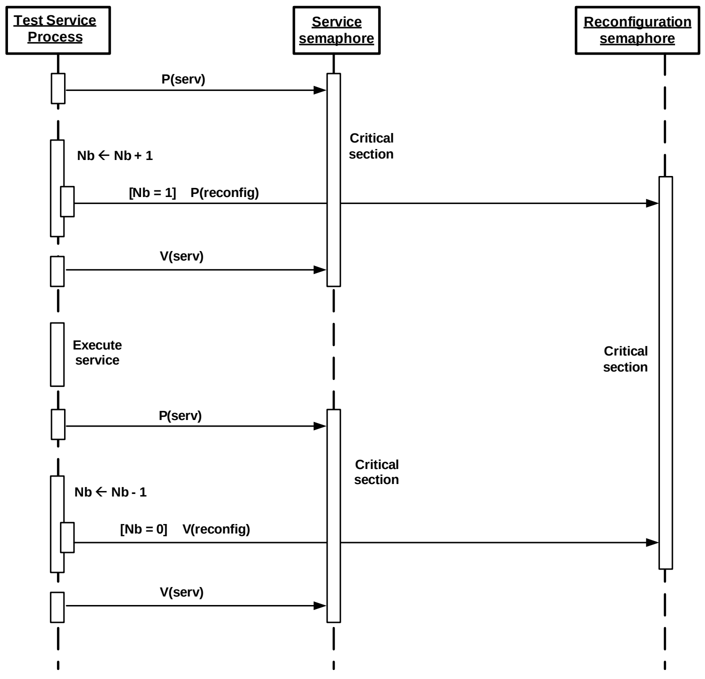

Figure 11: *The Service Process Scenario*

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 Control Component realizes the following instructions:

```
Algorithm 3: reconfigure a Control Component
begin
P(reconfig)
    execute the reconfiguration
V(reconfig)
end.
```

Running Example. *Let us take as example the Control Component $Elevate$ related to EnAS system. The agent needs to reconfigure this component which permits to launch the Elevate Reconfiguration Process. The Figure 12 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: (i) the operation $P(reconfig)$ leads to enter in critical section for Reconfiguration semaphore; (ii) the Elevate Reconfiguration Process executes the corresponding reconfiguration; (iii) the operation $V(reconfig)$ leads to liberate Reconfiguration semaphore from its critical section.*

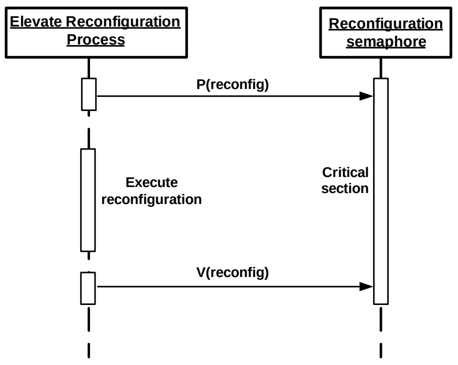

Figure 12: *The Reconfiguration Process Scenario*

6.2. 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 $P(reconfig)$ puts it into a waiting state. After the reconfiguration process which is running is terminated, the operation $V(reconfig)$ 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 $P(serv)$ by the process $P2$ leads it to a blocking state. When the process $P1$ terminates the test of the shared data Nb, the operation $V(serv)$ allows to launch the process waiting in the semaphore's queue.
- if $P1$ is executing its service, then the operation $P(serv)$ by the process $P2$ allows to execute normally.

Thus, this property is validated.

7. CONCLUSION

The paper deals with safety adaptive embedded control systems following the component-based approach. We propose an agent-based architecture to handle automatic reconfigurations under well-defined conditions by creating, deleting or updating Control Components to bring the system into safe and optimal behaviors. The architecture of the agent is modeled by nested state machines in which states correspond to other state machines in order to cover all possible reconfiguration forms. We propose conceptual models for the whole component-based architecture. The dynamic reconfiguration is ensured through a synchronization between service and reconfiguration processes to be applied to Control Components. We propose to use the semaphore concept for this synchronization such that we consider service processes as readers and reconfiguration processes as writers. All the results are applied in the paper to a particular benchmark production system.

8. REFERENCES

Crnkovic, I. and Larsson, M. (2002) *Building reliable component-based software systems*. Artech House.

J.P. Bodeveix, P. Dissaux, P. Farail, M. Filali, P. Gaufillet, F. Vernadat. (2006) *Behavioural descriptions in architecture description languages: Application to AADL*. In : European Congress on Embedded Real-Time Software (ERTS 2006), Toulouse, January 2006.

SZYPERSKI, C. and GRUNTZ, D. and MURER, S. (2002) *Component Software Beyond Object-Oriented Programming*. The Addison-Wesley Component Software Series.

Pratl, G. and Dietrich, D. and Hancke, G. and Penzhorn, W. (2007) *A New Model for Autonomous, Networked Control Systems*. IEEE Transactions on Industrial Informatics. Vol 3, No 1.

Gharbi, A. and Khalgui, M. and Hanisch, H-M. (2009) *Functional Safety of Component-based Embedded Control Systems*. 2nd IFAC Workshop on Dependable Control of Discrete Systems, Bari, Italy, June 2009.

Khalgui, M. and Hanisch, H-M. and Gharbi, A. (2009) *Model-Checking for the Functional Safety of Control Component-based Heterogeneous Embedded Systems*. 14th IEEE International conference on Emerging Technology and Factory Automation, Mallorca, Spain, September 2009.

Gharbi, A. and Khalgui, M. and Ben Ahmed, S. (2010) *Model Checking Optimization of Safe Control Embedded Components with Refinement*. 5th International conference on Design and Technology of Integrated Systems in Nanoscale Era, Hammamet, Tunisia, May 2010.

Gharbi, A. and Khalgui, M. and Ben Ahmed, S. (2010) *Optimal Model Checking of Safe Control Embedded Software Components*. 15th IEEE International Conference on Emerging Technologies and Factory Automation, Bilbao, Spain, September 2010.

Gharbi, A. and Khalgui, M. and Ben Ahmed, S. (2010) *Inter-Agents Communication Protocol for Distributed Reconfigurable Control Software Components*. The International Conference on Ambient Systems Networks and Technologies (ANT), Paris, France, November 2010.

Gharbi, A. and Khalgui, M. and Ben Ahmed, S. (2011) *Functional safety of discrete event systems*. First Workshop of Discrete Event Systems at Xidian University in China, February 2011.

Gharbi, A. and Gharsellaoui, H. and Khalgui, M. and Ben Ahmed, S. (2011) *Functional Safety of Distributed Embedded Control Systems*. Handbook of Research on Industrial Informatics and Manufacturing Intelligence: Innovations and Solutions / M. A. Khan, A. Q. Ansari (in press).

Rasche, A. and Polze, A. (2008) *ReDAC - Dynamic Reconfiguration of distributed component-based applications with cyclic dependencies*. Published in ISORC '08 Proceedings of the 2008 11th IEEE Symposium on Object Oriented Real-Time Distributed Computing, IEEE Computer Society Washington, DC, USA, 2008.

Rasche, A. and Schult, W. (2007) *Dynamic Updates of Graphical Components in the .NET Framework*. In Proceedings of SAKS07 Workshop.

Kramer, J. and Magee, J. (1990) *The Evolving Philosophers Problem: Dynamic Change Management*. IEEE Transactions on Software Engineering. Vol 16.

Papadopoulos, G. A. and Arbab, F. (2001) *Configuration and Dynamic Reconfiguration of Components Using the Coordination Paradigm*. Future Generation Computer Systems, Volume 17, Issue 8, June 2001, Pages 1023-1038.

D'Souza, D. and Wills, A. C. (1998) *Objects, Components and Frameworks: The Catalysis Approach*. Addison-Wesley.

Bidan, C. and Issarny, V. and Saridakis, T. and Zarras, A. (1998) *A Dynamic Reconfiguration Service for CORBA*. In CDS 98: Proceedings of the International Conference on Configurable Distributed Systems. IEEE Computer Society

Angelov, Ch. and Sierszecki, K. and Marian, N. (2005) *Design models for reusable and reconfigurable state machines*. L.T. Yang and All (Eds): EUC 2005, LNCS 3824, pp:152-163. International Federation for Information Processing.

Rooker, M. N. and Sunder, C. and Strasser, T. and Zoitl, A. and Hummer, O. and Ebenhofer, G. (2007) *Zero Downtime Reconfiguration of Distributed Automation Systems : The εCEDAC Approach*. Holonic and Multi-Agent Systems for Manufacturing Lecture Notes in Computer Science, 2007, Volume 4659/2007, 326-337.

Al-Safi, Y. and Vyatkin, V. (2007) *Ontology-based reconfiguration agent for intelligent mechatronic systems*. Robotics and Computer-Integrated Manufacturing, Volume 26, Issue 4, August 2010, Pages 381-391

Mati Y., Rezg N., Xie X. (2001) *Geometric Approach and Taboo Search for Scheduling Flexible Manufacturing Systems* Rapport de recherche (2001) 26 - http://hal.archives-ouvertes.fr/inria-00072489/fr/