A Decentralized Supervisory Control Approach For Distributed Adaptive Systems

Mohamed-Amine Belhaj Seboui⁽¹⁾, Nejib Ben Hadj-Alouane⁽¹⁾, Gwenaél Delaval⁽²⁾, Eric Rutten⁽²⁾, Moez Yeddes⁽¹⁾

⁽¹⁾National School of Computer Sciences, University of Manouba, Tunisia

⁽²⁾INRIA Rhône-Alpes, Grenoble, France

aminebs@yahoo.fr, nejib.bha@yahoo.com, gwenaél.delaval@inria.fr, eric.rutten@inria.fr, yeddes@yahoo.fr

In this paper, we propose a decentralized supervisory control approach for distributed adaptive systems. The adaptive systems are modeled, through their adaptation managers, as synchronous reactive systems using finite state machines with inputs/outputs and transition guards. When system complexity increases, critical properties cannot be verified intuitively. Therefore, formal and automated control synthesis methods are needed, for supervising the way the adaptation managers are coupled with the system, and for ensuring additional safety properties.

We propose the adaptation of existing techniques for discrete controller synthesis, to control decentralized systems that are composed of several subsystems spread across remote sites. Our approach consists of computing, for each site, abstracted behavior models for each distant subsystem. These behavior models will be coupled with the considered local subsystem, in order to compensate for the lack of information caused by the distribution constraints. Then we apply discrete controller synthesis on each site to obtain local controllers. These controllers, acting on their corresponding subsystems, will ensure that the global behavior property is respected.

Decentralized control synthesis techniques, Distributed Discrete event systems, Supervision and control, Reactive systems, Synchronous programming, Adaptive systems.

1. INTRODUCTION

Due to their proliferation in a great variety of environments, embedded systems must be able to perform self-reconfigurations in reaction to changes in their environments. This can be done through loading them with adaptation managers that are in charge of the computation of reconfiguration actions depending on the events and informations received from the external environments (Kephart and Chess 2003). These adaptation managers and their reconfiguration mechanisms can be considered as synchronous reactive systems (Halbwachs 1998). The synchronous aspect refers to the fact that the considered reactive systems interact to environments characterized by criticality, so actions must be executed instantly and deterministically in response to the events occurrences. Embedded systems are by nature safety-critical, due to their deep interaction with their embedding environments. In some applications (e.g. aeronautics), it is definitely not tolerable to fall-done or to pass over any fault. In addition to being adaptive, embedded system must be predictable.

To ensure safety, engineers, generally, use formal analysis and synthesis methods when designing critical systems. Specifically, Discrete Controller Synthesis (DCS) is a technique, based on Discrete-Event Systems (DES) framework (Ramadge and Wonham 1989) (Ben Hadj-Alouane et al. 1994), that allows for the construction of a controller respecting a given property. It presents the advantage of being constructive and gives practical solutions for solving the problem of adaptive component-based systems. Recently, this technique has been integrated within the component-based framework Fractal (Bruneton et al. 2006). In (Delaval and Rutten 2010), the authors used DCS for synthesizing reconfiguration controllers within Fractal-based environments.

The control theory of DES had been initiated by Ramadge and Wonham (1987, 1989). Within the synchronous approach, techniques and tools for classical DCS already exist. The classical DCS consists of synthesizing a controller that can observe all the system events and able to constrain the system as to respect a specified property. However, the requirements in terms of robustness, reliability

and fault tolerance impose the distribution and decentralization of embedded systems and, today, most practical real-life systems are composed of several subsystems spread across distant and remote sites. Therefore, a controller within a DCS environment is facing more complex systems and structural constraints (constraints of communication and visibility) which prevent it from observing the global system behavior and effectively controlling the distant subsystems.

In this paper we propose a method to control distributed adaptive systems that are modeled through their adaptation managers as synchronous reactive systems. Our model-based approach relies on the formalism of finite state machines with inputs/outputs and transition guards. We propose to take advantages of the existing DCS techniques and applying them on systems composed of several subsystems spread across remote sites. We propose to compute abstractions of behavior models of the distant subsystems that will be included within the local site of each considered subsystem. Then, we apply classical DCS on each local site, leading to a decentralized control solution, with a controller on each remote site. These controllers, associated with their corresponding subsystems, ensure that the global system respects a specified global property.

The paper is organized as follows. Section 2 introduces the model-based approach of synchronous reactive systems. In Section 3, we introduce the necessary notions from the control theory of discrete event systems. We present our approach for obtaining controlled decentralized systems in Section 4. An algorithm for computing the behavior models of the distant subsystems is given in Section 5. Finally, in section 6, we apply our approach on an example.

2. THE MODEL-BASED APPROACH OF SYNCHRONOUS REACTIVE SYSTEMS

The adaptation managers, implemented within the adaptive systems, are considered as synchronous reactive systems. These systems are characterized by their strong interaction with their environments. They continuously react to the demands of their environment through the manipulation of the input signals and the production of output signals. Within the synchronous approach (Halbwachs 1998), any system reaction is considered as an atomic event and is immediately propagated to its environment. All input values are read simultaneously and the computation of output values is done instantly. The notion of continuous time is replaced by the notion of discrete time which is guided by the input signals.

The synchronous reactive systems are, typically, modeled in terms of their events and states, as finite state machines. In this paper, we use Moore machines that are labeled with boolean formulas. Such machines are characterized by a 6-tuple $G = (Q, I, O, T, \lambda, q_0)$ consisting of the following:

- Q : is a finite set of states containing the initial state q_0 ;
- I : is a set of input boolean variables;
- O : is a set of output boolean variables;
- $T \subseteq Q \times f(I) \times Q$: is a set of transitions where, $f(I)$, is the set of boolean formulas on the variables in I ;
- $\lambda : Q \rightarrow \mathbb{B}^{|O|}$: is the output function that defines the values of the output variables associated with each state.

In the following, we will refer to such machines simply as automaton. The set of transitions, T , defines the possible transitions between the states of the automata. Each transition, $T_i \in T$, is a triplet (s, f_i, s') where s is the starting-state, s' the next-state and $f_i \in f(I)$ is a boolean formula on the variables in I acting as a guard on the transition. We denote such a transition by $T_i : s \xrightarrow{f_i} s'$. A transition can be executed only if its associated formula is satisfied by the values of the input variables (i.e. its logical value equals to *true*). We define the valuation of a formula, noted $\mathcal{V} : f(I) \rightarrow \{true, false\}$, which computes its logical value, corresponding to the current interpretation of the values of the input variables. The following example illustrates how the above automata are well suited for the modeling of self-reconfiguring adaptive systems.

Example 1 We consider the example of adaptive system illustrated in Figure 1. The given automaton

Figure 1: Connection System in a Cell-Phone.

represents the component in charge of the Internet connexion in a cell-phone. This system is able to connect to the Internet either through the telephone network connexion, or through a wireless

network connection. In order to ensure an optimal performance in terms of quality and cost, this system is equipped with adaptation mechanisms to perform self-reconfiguration according to current external network environment within which exists the cell-phone. So, on the detection of a wireless network in the vicinity, the system, initially connected to the telephone network, will disconnect from it and connect to the wireless network that offers a higher throughput and a lower cost. Similarly, when the phone is out of the reach of the wireless network, the connection system will do self-reconfiguration and reconnect to the telephone network. The automaton modeling this system has three states modeling different configurations: *Idle* (not connected), *Wifi* (connected to the wireless network) and *Tel* (connected to telephone network). The input variables are w , which indicates that a wireless network is detected, and t , which indicates that a telephone network is detected. A set of transitions, labeled with boolean formulas on $\{w, t\}$, give the possible transitions between these states. There is just one output variable, *connected*, associated to each state, indicating whether the system is connected or not. $\square$

Systems modeled by Moore machines present a so-called equational view, illustrated in Figure 2. It is formally defined as follows:

- a state X , presented through a set of state variables;
- a finite set of input variables I ;
- a finite set of output variables O ;
- an output function $out : \mathbb{B}^{|X|} \rightarrow \mathbb{B}^{|O|}$;
- a transition function $g : \mathbb{B}^{|I|} \times \mathbb{B}^{|X|} \rightarrow \mathbb{B}^{|X|}$.

Figure 2: Equational View of a Reactive System.

3. THE CONTROL THEORY OF DISCRETE EVENT SYSTEMS

The control theory of discrete event systems (DES) had been initiated by Ramadge and Wonham (1987, 1989). Their approach is based on the modeling systems behavior by languages, which are generated by finite state machines. The alphabet

of the language is the set of events observed by the system. The objective of controlling such a system is to reduce its behavior to a desired sublanguage that respects a specified property. The control of a system, whose behavior is modeled by a language L and given desired behavior modeled by a sublanguage $K \subseteq L$, consists of computing a supervisor that will dynamically enable/disable some events, said the controllable events, to keep the system within the desired behavior. Ramadge and Wonham proved the existence of the maximally permissive control and approaches have been given for computing it (Ben Hadj-Alouane et al. 1994). Their approach has been applied on other models as Petri Nets (Krogh and Holloway 1991).

An other approach from control theory (Balemi 1992), called input/output control, considers that the events received from the environment can not be disabled. So, all these events are considered as uncontrollable variables. In this case, the system is controlled by providing a set of control commands, called controllable variables, to enforce a given property. Within the framework of the synchronous reactive systems, the input/output control is used. Considering the equational view of a reactive system, given in Figure 2, the discrete controller synthesis (DCS) consists of computing a function, $h : \mathbb{B}^{|I_u|} \times X \rightarrow \mathbb{B}^{|I_c|}$, that will compute the values of the controllable variables according to the values of the uncontrollable variables and the current system state. Figure 3, illustrates the controlled system. The set of input variables is divided into two disjoint subsets, I_u , the uncontrollable variables, received from the environment and I_c , the controllable variables, computed by h .

Figure 3: Control of a Reactive System.

Techniques for DCS are already available allowing to resolve the classical control problem of DES (i.e. with full-observation of the system behavior). Many works, such as (Lin and Wonham 1988b), (Cieslak et al. 1988) and (Ben Hadj-Alouane et al. 1996), proposed solutions for the problem of control under partial observation which is encountered when the controller cannot observe all of the system events.

Other works had proposed to decompose the control objective into a conjunction of objectives and to synthesize a supervisor for each decomposed

objective. This is the case of modular supervision presented in (Ramadge and Wonham 1988) and decentralized supervision presented in (Lin and Wonham 1988a), (Rudie and Wonham 1992) and (Barrett and Lafortune 2000). In (Willner and Heymann 1991), the authors introduce the decentralized supervisory control of concurrent DESs which are modeled by a composition of subsystems and, in (Abdelwahed and Wonham 2002), the authors extend the notion of decentralized supervisory control to a class of systems, called Interacting DESs. These cited works are rooted, principally, within the framework of the language approach.

Within the synchronous approach, in classical DCS, which is concerned with the control of fully-observed DESs, tools have been defined and implemented with the context of synchronous languages, as SIGALI (Marchand et al. 2000). Then, techniques for the control of modular systems had been proposed. In (Delaval et al. 2010), the authors present a method for the modular application of DCS to synchronous programs and an implementation is given with the context of the BZR language, which is presented, also, in (Delaval et al. 2010). With our framework, we model our systems by automata with the synchronous language BZR which can be interfaced with SIGALI in order to apply DCS. SIGALI, however, requires the use of a global view of the system and can not be applied, directly, to decentralized systems. Therefore we propose, in the next section, a new method for the control of decentralized adaptive systems.

4. OUR APPROACH FOR THE CONTROL OF DISTRIBUTED ADAPTIVE SYSTEMS

Classical DCS applied to a system S computes a controller C . The controlled system, S_c , obtained from the synchronous connection of S and C , ($S_c = S \parallel C$), respects a property specified in the contract of the DCS. The systems that we consider are composed of several subsystems spread across remote sites. There are many constraints that prevent classical DCS from computing a controller to such systems like the incapacity to observe all the events that occur at the different subsystems specially when the DCS has to enforce a property that depend on the behavior of many of the distant subsystems, at the same time, like mutual exclusion when using shared resources. An other constraint is due to the communication environment: when using a centralized controller each subsystem must communicate with it in order to get the appropriate values of the controllable variable, which can be impossible. The solution consists of computing local controllers that will be associated with each of the

corresponding subsystem. Every local controller will compute the values of controllable variables of its correspondent subsystem in order to ensure that the behavior of the global system respects a specified global property.

Considering a system composed of n subsystems ($S = S_1 \parallel \dots \parallel S_n$) the controlled system, illustrated by Figure 4, can be described as follows:

$$S_c = (S_1 \parallel C_1) \parallel (S_2 \parallel C_2) \parallel \dots \parallel (S_n \parallel C_n).$$

Figure 4: Control of a Decentralized System through Local Controllers.

When the global property, to be ensured, can be decoupled into local properties that, only, involve the corresponding local subsystem behavior (e.g a set of prohibited states), the problem of decentralized DCS is simple and local controllers can be computed, directly, for each subsystem to enforce its local property. But, in most cases, the local properties involve the behavior of several distant subsystems (e.g. exclusive use of shared resources). Besides, the available information about the distant subsystems behaviors, that the local controllers can observe, consists generally on a set of input and output variables, called the observable variables, that had been sent by the other subsystems, or are available due to their global aspect (e.g. temperature). This is not sufficient, however, to the controller to predict the future behavior of the distant subsystem. Therefore, our solution consists on computing abstracted behavior models of the distant subsystems that will model their behaviors and will use the observable variables to predict, as possible, their actions. Then, we will synchronously associate the local subsystem and the behavior models of the distant subsystems and we apply DCS on the composition in order to obtain a local controller. Figure 5 shows the composition, in a local site ($site_i$), of the local subsystem (S_i) with the abstracted behavior models of the subsystems of all other sites ($M_1 \dots M_{i-1} M_{i+1} \dots M_n$).

Each abstracted behavior model must take advantages from all available information in order to reflect a behavior that is the closest to the real subsystem.

Figure 5: Composition of the local subsystem and the abstracted behavior models of the distant subsystems.

The proportion of the observable variables affect the abstraction level of the behavior model. The more observable variables there are, the lower the abstraction level of the computed behavior model will be (i.e the behavior model will be more detailed). There is not a fixed method to compute the behavior models. In general case, the behavior models must reflect the controllability aspect of the distant subsystem and predict their behavior according to the values of the observable variables. In the next section, we propose an algorithm to compute the behavior models that can be used in general cases without a restriction on the property to be ensured.

5. ALGORITHM FOR THE COMPUTATION OF THE BEHAVIOR MODELS

5.1. Preliminaries

The problem statement of behavior models computation consists of: Given a distant subsystem modeled by an automaton, a global property as a boolean formula and a set of observable variables in a local site, we must compute an automaton, modeling the behavior of the distant subsystem and allowing to be coupled to the local subsystem in order to synthesize a controller (by DCS) that will ensure a local property derived from the global one.

Our idea for the behavior models computation consists of computing an automaton where every state symbolizes a set of the distant subsystem states that contains all the states where the subsystem is suspected to be into. Before introducing the principal steps of the proposed algorithm, we give, in this subsection, some preliminaries.

Local Property Determination

The global property is a boolean formula on the output signals of the subsystems. It can be a collection of several independent properties that the controller must verify all together. Since the local controller is, only, in charge of the control of its local subsystem, it is reasonable to define a local property that is concerned only with the control of the local subsystem. For this to be carried out, we convert the global property into its equivalent conjunctive normal form which divides it into a

conjunction of clauses¹. Then, the local property will be the conjunction of the clauses that include at least one of the local subsystem output signals. Doing this, the local controller will not need to observe the behavior of the distant subsystems that are not implicated in this local property. For each site we will include only the abstracted behavior models of the concerned subsystems (i.e. the subsystems that have some output variables implicated in the local property). In addition, the output signals of the distant subsystems will be replaced, within the local property, by equivalent output signals assigned to the computed abstracted behavior models.

Controllability Aspects

For a local controller, each signal, coming from a distant site, is considered as uncontrollable variable. Since a local controller must observe the distant subsystem behavior in order to control its local subsystem, it must also observe whether a transition is controllable or not and whether a critical behavior of the distant subsystem can be restricted or not. This can be illustrated by the case of exclusive use of a shared resource: when the distant subsystem is out of the critical area, the controller can allow its local subsystem to move to its critical area only if it is sure that the distant subsystem can be prevented from moving to its critical area. Therefore, to reproduce the controllability aspect of a subsystem behavior, we will replicate the controllable variables of the real subsystem in its behavior model. This way, the local controller will consider that it is controlling the behavior model which is a replication of the real control applied to the distant subsystem.

Deterministic Aspect of the Considered Automata

In the case of the Moore machines labeled with boolean formulas, an automaton is said deterministic, if for each state $s \in Q$ and for each pair of transitions, $T_1: s \xrightarrow{f_1} s'$, and, $T_2: s \xrightarrow{f_2} s''$, we, always, have $\mathcal{V}(f_1 \wedge f_2) = false$. Nevertheless, in practical case, it is not necessary to integrate for each transition the negation of the other transitions formulas, because the system evaluates transitions formulas according to their order of implementation. This implies that each formula implements implicitly the negation of the transitions formulas that precede it. Within our algorithm, we will take this into consideration. For the illustration of examples, when there is ambiguity, we will label the transitions by an integer i which will indicate the order of implementation. In this way, a transition T_j is implemented before T_k , only if $j < k$ and we will say that T_j precede T_k .

¹A clause is a disjunction of literals, and a literal is either a variable or a negated variable

5.2. Principal Steps

The principal steps of the algorithm that we propose are presented by Algorithm 1.

Algorithm 1: getModel

Input: A : Initial Automaton; $Signals$: the set of available signals; P : Local property.
Output: A_{mod} : The abstracted behavior model automaton.

```

begin
  (1) Compute the intermediate automaton;
  (2) Compute the behavior model  $A_{mod}$ ;
  (3) Assign output variables to  $A_{mod}$ ;
  return  $A_{mod}$ ;
end

```

The inputs of the algorithm are: 1) A : the automaton of a distant subsystem. 2) $Signals$: the set of observable signals in the local site. 3) P : the property assigned to the local subsystem. Given these, the algorithm will compute the abstracted behavior model automaton of A that will be included in the considered site. Step (1) consists of computing an intermediate automaton, where we make some modifications to the initial automaton, such as eliminating the unobservable input signals and incorporating the observable output signals. In step (2), we compute the behavior model. Finally, in step (3), we assign the output signals to the computed model. These steps are detailed in the following.

The computation of the intermediate automaton is given by the Algorithm 2.

Algorithm 2: getIntermediateAutomaton

Input: A : Initial automaton; $Signals$: the set of observable signals.
Output: A_{int} : Intermediate automaton.

```

begin
  (1) Eliminate unobservable signals and mark the partial transitions;
  (2) Incorporate the observable output signals into the conditions of the transitions;
  (3) foreach state  $S_i$  in  $A$  do
    Compute the intermediate state of  $S_i$  and add it to  $A_{int}$ ;
  end
  return  $A_{int}$ ;
end

```

In step (1), we begin by eliminating the unobservable uncontrollable signals from the formulas of the transitions. For example, if we consider a formula, $f = (a \wedge b) \vee (\neg a \wedge c)$, and we assume that the variable a is unobservable, then f will be replaced

by $f' = (b) \vee (c)$. In general case, the elimination of a signal o_i from a formula f_j can be done by: first, transforming f_j to its disjunctive normal form and, then, replacing o_i and its negation $\neg o_i$ by *true*.

The elimination of the unobservable uncontrollable signals leads to two types of transitions: 1) **partial transition**: which is a special transition where some uncontrollable signals that are involved in the transition formula are unobservable and are removed. With the remaining part of the formula (partial formula) we can not assert if this transition can be executed or not. 2) **non-partial transition**: which is a normal transition, where all uncontrollable signals involved in the transition formula are observable. This transition can be executed only if its formula is satisfied by the input values.

Figure 6 shows an example dealing with eliminating unobservable signals and marking the partial transitions issued from a state $E1$. The original system is represented on the left. In this example we consider that the observable signals are b and e , and that c_1 is a controllable variable. The resulting system is presented on the right of the figure. The partial transitions are marked with (P) and the non-partial transitions are marked with (NP).

Figure 6: Elimination of Unobservable Signals and Marking the Partial Transitions.

In step (2) of Algorithm 2, following unobservable signals elimination, we will incorporate the observable output signals, that may have been transmitted from the distant subsystem to the local one, into the transitions formulas. This will give some advantages because output signals, when their values change, can give idea of the current state of the system. Specially when for a given state, there is only one state, from the set of the states that follows, where the value of an output signal is not the same. In this case, if the output value change then automatically we conclude that the system moved to this state.

The step (3) of Algorithm 2, consists on computing an intermediate state for each state of the initial automaton. We assign to each intermediate state new transitions issued from combinations of the partial and non-partial transitions of the initial state. We introduce the notion of transitions conjunction as following: a conjunction of transitions $T_1, T_2, \dots, T_n$ issued from the same state s and where each

transition is denoted $T_i : s \xrightarrow{f_i} s_i$, consists on creating a new transition T' labeled with the logical conjunction of the transitions formulas and where its next-state is symbolizing all their next-states ($T' : s \xrightarrow{f'=f_1 \wedge \dots \wedge f_n} s'$, where s' is symbolizing $s_1, s_2, \dots, s_n$). Algorithm 3 presents the function in charge of the intermediate state computation.

Algorithm 3: getIntermediateState

Input: S : Initial state
Output: S_{int} : Intermediate state
Var: Tab : Array of transitions
begin
 foreach transition T_i from S **do**
 if T_i is partial **then**
 add T_i to Tab ;
 else if size of $Tab \neq 0$ **then**
 Add to S_{int} transitions issued from the
 conjunction of T_i with all possible
 combinations of transitions in Tab ;
 else
 Add T_i to S_{int} ;
 end
 end
 if size of $Tab \neq 0$ **then**
 Add to S_{int} transitions issued from the
 conjunction of all possible combinations of
 transitions in Tab ;
 end
 return S_{int} ;
end

The computation of the intermediate states is done by scanning transitions with respect to the order of their implementation and is based on this two rules: 1) If a non-partial transition T_i is encountered, Then new transitions are built by the conjunctions of T_i with all possible combinations of the partial transitions that precede T_i . 2) When all non-partial transitions are treated, new transitions will be built by the conjunctions of all possible combinations of the partial transitions and by adding the starting-state to the symbolized states of the next-state of the new transitions. This is due to that even if the partial formula is satisfied, we can not assert whether the system moved to the next state, or remained on the starting-state.

Figure 7 shows the computed intermediate state corresponding to the state $E1$ of the example given in figure 6. To describe this example, we will refer to the transitions of the initial state, presented in figure 6, by T^S and the transitions of the computed intermediary state, presented in figure 7, by T^I . According to the algorithm, when the non-partial transition $T_2^S : E1 \xrightarrow{b} E3$ is encountered, the conjunction of T_2^S with $T_1^S : E1 \xrightarrow{c_1} E2$ (the

Figure 7: intermediate state

only partial transition that precede T_2^S) lead to the transition $T_1^I : E1 \xrightarrow{b \wedge c_1} E3E2$. Then, $T_2^I : E1 \xrightarrow{b \wedge \neg c_1} E3$, is labeled with the conjunction of the formula of T_2^S and the negation of the formula of T_1^S , and, obviously, the next-state symbolizes only the next state of T_2^S . Next, there is no more non-partial transitions. So, the algorithm will compute the transition $T_3^I : E1 \xrightarrow{c_1 \wedge e} E1E2E4$ by the conjunction of all the partial transitions, which are T_1^S and $T_3^S : E1 \xrightarrow{e} E4$, and with adding $E1$ to the symbolized states of the next-state. Then, in $T_4^I : E1 \xrightarrow{c_1 \wedge \neg e} E1E2$, we consider the negation of the formula of T_3^S , and, in $T_5^I : E1 \xrightarrow{\neg c_1 \wedge e} E1E4$, we consider the negation of the formula of T_1^S . In $T_6^I : E1 \xrightarrow{\neg c_1 \wedge \neg e} E1$, we consider their both negations and the next-state of T_6^I symbolize only the starting-state.

After computing the intermediate automaton, the step (2) of Algorithm 1 consists on computing the automaton of the abstracted behavior model. Algorithm 4 present the function in charge of this.

Algorithm 4: getModelAutomaton

Input: A_{int} : Intermediate automaton
Output: A_{mod} : The behavior model automaton
begin
 Add the first-state of A_{int} to A_{mod} ;
 foreach new state S_i added to A_{mod} **do**
 if number of symbolized states by $S_i > 1$ **then**
 Build the transitions issued from the
 multiple state S_i ;
 end
 foreach transition T_i outgoing from S_i **do**
 if A_{mod} don't already contain the
 next-state of T_i **then**
 Add the next-state of T_i to A_{mod} ;
 end
 end
 end
 return A_{mod} ;
end

The abstracted behavior model will have, as starting state, the starting state of the intermediate automaton with its new computed transitions. Then

we will add to the model, progressively, each next state of the transitions issued from the initial state. Some states, that we call multiple states, are newly computed by the intermediate automaton and are symbolizing more than one of the original states. The transitions issued from this states must be computed by combining the transitions issued from each symbolized state. For each new built transition, we will conjunct one transition from each state, so, the transition formula will be the conjunction of these transitions formulas and its next-state will symbolize the next-states of the combined transitions. An example of computing a multiple state is given by figure 8. The multiple state $A1B1$, illustrated in figure 8.b, is symbolizing two states $A1$ and $B1$ illustrated in figure 8.a.

Figure 8: Computation of a multiple state transitions

The last step of Algorithm 1, consists of assigning the output variables to the computed behavior model. The procedure, in charge of this, is presented by Algorithm 5.

In the first part, the procedure will replace, within the local property, the output variables of the initial automaton by equivalent output variables that will be assigned to the behavior model. In the second part, the procedure will assign, for each state, the values of these equivalent output variables. This is done by considering the values of the original variable in each symbolized state. For each state S_i of the behavior model, if for all the symbolized states by S_i the value of o_i^S , the original output variable, is the same, then o_i^M , the equivalent output variable assigned to the behavior model, will get the value of o_i^S (or $\neg o_i^S$ if it symbolizes its negation). When the value of o_i^S is not the same in all the symbolized states, the value *false* will be given to o_i^M . This amounts to tell the controller that in this state the value of o_i^M is uncertain then consider that it is *false* and ensure the remaining part of the property. This can be seen as if we consider the worst case.

Algorithm 5: assignOutputsToModel

Input: A_{mod} : The behavior model automaton; A : Initial Automaton; P : Local property
Output: Assign outputs to the model
begin
 foreach output o_i^S involved in the local property **do**
 Replace o_i^S by o_i^M in the local property;
 Replace $\neg o_i^S$ by $\neg o_i^M$ in the local property and mark it as a negation;
 Add o_i^M and $\neg o_i^M$ to the outputs set of A_{mod} ;
 end
 foreach state S_i in A_{mod} **do**
 foreach output o_i^M in A_{mod} **do**
 if the value of o_i^S is the same in all the states symbolized by S_i **then**
 if o_i^M is a negation **then**
 $o_i^M \leftarrow \neg o_i^S$;
 else
 $o_i^M \leftarrow o_i^S$;
 end
 else
 $o_i^M \leftarrow False$;
 end
 end
 end
end

6. APPLICATION EXAMPLE

We consider a system composed of two distant subsystems S_1 and S_2 presented in figure 9.

Figure 9: System modeling two distributed process

The subsystem S_1 is modeling a process with three states: Idle I , Active A and Error E . The uncontrollable input variables of this subsystem are d , f and e . c is a controllable variable. It has two output variables: act^{S_1} which indicates that the subsystem is in the active state, and err^{S_1} , which indicates that the subsystem is in the error state. For clarity, we represent the output variables only where their value is equal to *true* and to avoid confusion we will, for each output variable, indicate its corresponding subsystem (e.g. a variable o_i that belongs to S_1 is denoted by $o_i^{S_1}$).

The subsystem S_2 is modeling a process with three tasks. Initially in its idle state I , S_2 can move to its states A_1 , A_2 and A_3 which are modeling the three tasks. Its uncontrollable input variables are d_1 , d_2 and f_2 and it has two controllable variables that are c_1 and c_2 . Its output variables are $act_1^{S_2}$, which its value is equal to *true* only in the state A_1 , and $idle^{S_2}$ which indicates that the subsystem is in its idle state.

Concerning the property to ensure, we suppose that the states A of S_1 and A_1 of S_2 are conflicting (e.g. due to the use of a shared resource) and that when the subsystem S_1 move to the error state E , the subsystem S_2 must return to its idle state I . We conclude the following property that reflects these two constraints and that will be the objective of the control:

$$P = (\neg act^{S_1} \vee \neg act_1^{S_2}) \wedge (\neg err^{S_1} \vee idle^{S_2})$$

To control this system, we will apply our approach which consists, in this case, on computing a behavior model, M_1 , of S_1 and a behavior model, M_2 , of S_2 . Then we will incorporate M_2 in the local site, $Site_1$, of S_1 and M_1 in the local site, $Site_2$, of S_2 . Finally, we will apply DCS on each site to obtain local controllers that will enforce P . Figure 10 shows the considered distributed architecture of the system. In this architecture, we suppose that, in $Site_2$, the observable variables of S_1 are e and f and, in $Site_1$, we can observe only the output variable $act_1^{S_2}$ of S_2 . Then, M_1 will have as uncontrollable input variables e and f and will have a controllable variable c which is a replication of the one of S_1 . M_2 will have, as uncontrollable input variables, $act_1^{S_2}$ and, as controllable variables, c_1 and c_2 .

The output variables assigned to the behavior models depend of the output variables involved in the property. Then, according to the algorithm, M_1 will have as output variables $nact^{M_1}$, equivalent to the negation of act^{S_1} , and $nerr^{M_1}$, equivalent to the negation of err^{S_1} . So, the local property assigned to $Site_2$ will be:

$$P_2 = (nact^{M_1} \vee \neg act_1^{S_2}) \wedge (nerr^{M_1} \vee idle^{S_2})$$

Figure 10: Distributed Architecture of the Considered System

By the same, M_2 will have as output variables $nact_1^{M_2}$, equivalent to the negation of $act_1^{S_2}$, and $idle^{M_2}$, equivalent to $idle^{S_2}$. So, the local property assigned to $Site_1$ will be:

$$P_1 = (\neg act^{S_1} \vee nact_1^{M_2}) \wedge (\neg err^{S_1} \vee idle^{M_2})$$

Figure 11 presents M_1 , the automaton of the behavior model of S_1 computed according to the algorithm.

Figure 11: The Behavior Model of S_1

Figure 12 presents M_2 , the automaton of the behavior model of S_2 computed according to the algorithm.

After computing M_1 and M_2 , these behavior models will be incorporated in their correspondent sites: M_1 in $site_2$ and M_2 in $site_1$. Then we apply DCS on these two sites which leads to two controllers C_1 and C_2 . Figure 13 illustrates the architecture of the controlled distributed system issued from the composition of each site with its correspondent controller.

7. CONCLUSION

In this paper, we presented a method for controlling distributed adaptive systems. Our approach consists of computing, for each local subsystem, abstracted behaviors models of the distant subsystems. Then we apply DCS on each composition of the local subsystem together with the necessary abstracted behavior models to obtain a local controller for

Figure 12: The Behavior Model of S_2

Figure 13: The Controlled Distributed System

each site. These local controllers will act on the subsystems to ensure a global property. We proposed an algorithm to compute these abstracted behavior models. We then applied our approach on several examples composed of several subsystems, and we verified our results by simulating the system in a centralized environment and applying model-checking which approved that the composition of the subsystems with their corresponding controllers respect the specified property. However, we are looking forward to formally proving our results and providing qualitative measures for our decentralized controllers in our future work.

8. REFERENCES

- S. Abdelwahed and W.M. Wonham (2002). Supervisory control of interacting discrete event systems. In Decision and Control, 2002, Proceedings of the 41st IEEE Conference on, volume 2, pages 1175-1180 vol.2.
- S. Balemi (1992). Control of Discrete Event Systems : Theory and Application. PhD thesis, Automatic Control Lab., Swiss Federal Inst. of Technology (ETH), Zurich, Switzerland, May 1992.
- G. Barrett and S. Lafortune (2000). Decentralized supervisory control with communicating controllers. Automatic Control, IEEE Transactions on, 45(9) :1620-1638.

Nejib Ben Hadj-Alouane, Stéphane Lafortune and Feng Lin (1994). Variable lookahead supervisory control with state information. *IEEE Trans. Automatic Control* 39(12): 2398-2410.

Nejib Ben Hadj-Alouane, Stéphane Lafortune and Feng Lin (1996). Centralized and distributed algorithms for on-line synthesis of maximal control policies under partial observation. *Discrete Event Dynamic Systems* 6 (4): 379-427.

E. Bruneton, T. Coupaye, M. Leclercq, V. Quema, and J.-B. Stefani (2006). The fractal component model and its support in java. *Software Practice and Experience (SP&E)*, 36(11-12).

R. Cieslak, C. Desclaux, A.S. Fawaz, and P. Varaiya (1988). Supervisory control of discrete-event processes with partial observations. *Automatic Control*, IEEE Transactions on, 33(3) :249-260.

Gwenaél Delaval, Hervé Marchand and Eric Rutten (2010). Contracts for modular discrete controller synthesis. In ACM International Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES 2010), Stockholm, Sweden, April 2010.

Gwenaél Delaval and Eric Rutten (2010). Reactive model-based control of reconfiguration in the fractal component-based model. In Proceedings of the 13th International Symposium on Component Based Software Engineering (CBSE), Prague, Czech Republic, 23-25, June 2010.

N. Halbwachs (1998). Synchronous programming of reactive systems, a tutorial and commented bibliography. In Tenth International Conference on Computer-Aided Verification, CAV98, Vancouver (B.C.), June 1998. LNCS 1427, Springer Verlag.

J. O. Kephart and D. M. Chess (2003). The vision of autonomic computing. *IEEE Computer*, 36(1):41-50.

B. H. Krogh and L. E. Holloway (1991). Synthesis of feedback control logic for discrete manufacturing systems. *Automatica*, 27(4) :641-651.

F. Lin and W. M. Wonham (1988a). Decentralized supervisory control of discrete-event systems. *Inf. Sci.*, 44(3) :199-224.

F. Lin and W. M. Wonham (1988b). On observability of discrete-event systems. *Inf. Sci.*, 44(3) :173-198.

H. Marchand, P. Bournai, M. LeBorgne, and P. LeGuernic (2000). Synthesis of discrete-event controllers based on the signal environment. In *Discrete Event Dynamic System : Theory and Applications*, pages 325-346.

K. Rudie and W.M. Wonham (1992). Think globally, act locally : decentralized supervisory control. Automatic Control, IEEE Transactions on, 37(11) :1692-1708.

P. J. Ramadge and W. M. Wonham (1987).
Supervisory control of a class of discrete event

processes. SIAM J. Control Optim., 25 :206-null, 1987.

P. J. Ramadge and W. M. Wonham (1988). Modular supervisory control of discrete-event systems. Mathematics of Control, Signals, and Systems (MCSS), 1 :13-30.

P.J. Ramadge and W.M. Wonham (1989). The control of discrete event systems. Proceedings of the IEEE, 77(1) :81-98.

Y. Willner and M. Heymann (1991). Supervisory control of concurrent discrete-event systems. International Journal of Control, vol. 54, no. 5, pp. 1143-1169, 1991.