CONTROL SYNTHESIS FOR RECONFIGURABLE DISTRIBUTED SYSTEMS WITH APPLICATIONS IN MANUFACTURING

Ardavan Amini
Peng Zhao
Mohsen A. Jafari

Rutgers, The State University Of New Jersey
Department Of Industrial And Systems Engineering

Abstract: We present a framework for control synthesis of reconfigurable distributed agent systems, where each agent includes its own set of basic functions and local rules. There are also global rules, which govern the interaction and communication between these agents. Agents are logically defined by a control synthesizer, communicator, and executer. The control synthesis is capable of detecting and avoiding single level deadlocks at execution level. We illustrate the concept through an example.

Keywords: Reconfigurable control systems – Intelligent agents – Distributed systems – Synthesis – Agent communication – Deadlock – IEC 61499.

1. INTRODUCTION

This work is inspired by the fact that many man-machine systems, manufacturing and business alike, require fast response to the changes in their environment. In manufacturing systems, changes in demands, part designs and machine failures all require system reconfiguration at various levels of system control and communication hierarchy. In business systems, changes and fluctuations in market dynamics, business relationships and protocols between partnering organizations require reconfiguration in some or all levels of business functions.

In centralized systems, the changes are often dictated from the top to the units and functions at the lower levels. Although this may sound efficient and more practical, the fact of the matter is that more and more real systems in many application areas are becoming distributed in nature. In distributed systems, various units or functions must determine their own way of responding to the changes in their environment, and communicate these changes to the other units and functions. Here we will only focus on logical aspects of system reconfiguration, that is, at system control level. To be reconfigurable at system control level, it is necessary that the control logic is model-based and furthermore, the underlying processes and rules are separated from each other. The boundary between "processes" and "rules" is rather vague. In discrete-event control theoretic framework, Ramadge and Wonham (1987a, b) tossed this idea for the first time, and built an algorithmic solution for controller synthesis. We will follow their line of thought in terms of separation of processes and rules, so that rules are flexible and changeable, and can be changed while processes are fixed. Therefore, depending on the level of control, the functions performed by a machine or tasks performed by a human agent are part of fixed processes, whereas, "process plan" describing flow of materials, or the

sequence by which a human agent must do his/her tasks are rules. It goes without saying that the terms "flexible," "changeable" or "fixed" are relative within a control context.

The control synthesis for centralized systems have been extensively discussed and analyzed within the discrete-event system control-theoretic framework. Classical methods include techniques by Petri Nets (Zhou and DeCesare, 1993), supervisory control theory developed by Ramadge and Wonham (1987a, b), control synthesis via condition/event systems (Krogh and Kowalewski, 1996, Hanisch and Rausch, 1995) and time transition models (Ostroff, 1989). These works are model-based, synthesizing the complete controller from specifications provided as input. Should the rules or the underlying processes change, the whole synthesis algorithm must be run again in order to synthesize the control model again. This is in contrast to our approach where we synthesize control actions only when needed. The system will have a memory of its previously synthesized control actions, which can be overridden and changed if necessary. Our main inspiration for this approach comes from the way humans operate. As a human, we are never fully pre-programmed for all the tasks that we do in our lives. We rather carry with ourselves a set of rules (knowledge) and a set of basic functions (skills), which of course can be improved and changed by training and learning. When we are given a task (which we have not previously seen), we basically develop our own set of actions and procedures in order to accomplish this task. Furthermore, to complete this task, we may need to establish communication with other human or machine agents in our work environment. If we were given a task, which we have seen before, we may still need to reestablish our control actions, as the existing environmental conditions can be significantly different than what we had experienced in the past.

Our modeling framework encompasses the following major characteristics: Agents can be providers and/or requesters. Each agent embeds in it a set of basic functions and a set of local rules or control specifications. Agent actions are triggered by internal or external events (described by flags). Each such event or flag is associated with a goal, which must be achieved by the agent. Depending on its current condition and this goal, the agent internally synthesizes its set of control actions in order to accomplish the goal. It is very possible that several agents (providers) are simultaneously responding to the same event triggered by a requester agent. In such a case, the requester would select that provider which provides the least-cost solution.

The agents have different ways of computing their control synthesis solutions. In the simplest form, an agent may just adopt the first feasible solution, which defines the set of control actions (tasks) that must be taken to reach the goal from the current state. In a more complex form, an optimal solution may be sought, taking into account current state of the agent (provider), the ongoing tasks, and other tasks, which must be accomplished. Yet, in another scenario, the provider agent may face competition from other provider agents. Finally, solution may be in the face of uncertainties associated with failures or drop offs from other agents. Here we will only focus on the simplest solution where the first feasible solution obtained is priced and communicated to the requester. It is of course possible that in some cases, no feasible solution is obtained due to conditions such as deadlocks, etc.

2. METHODOLOGY

2.1 Definitions

Agent: An agent is a finite set $\Phi = (S, F, R, f, A, In)$, where: $S = \{s_1, s_2, ..., s_m\}$ is the set of states,
$F = \{f_1, f_2, ..., f_n\}$ is the set of basic functions,
$R = \{r_1, r_2, ..., r_s\}$ is the set of rules,
$f = \{\varphi_1, \varphi_2, ..., \varphi_r\}$ is the set of flags,
$A = \{a_1, a_2, ..., a_t\}$ is the set of attributes and
$In = \{i_1, i_2, ..., i_w\}$ is the set of initiators.

Entity: An entity E is an object that is processed and manipulated by the agents and can be shown by the tuple $E = (PP_i, j, k)$, where PP_i is the processing sequence id associated with the entity E, j is an instance id of the entity E and k is the current stage in the processing sequence.

Processing Sequence: is an ordered set of agents where PP_i = {All the agents Φ_i which must be visited in a sequential order by the entity E_i}

State: Each agent is associated with a state defined by an n-tuple, where n is a finite number. State of an agent changes upon execution of a basic function or occurrence of an external event.

Basic Functions: The primitive capabilities of an agent; they are pre-defined by the system and cannot be changed by the developer. Every function has a set of inputs and outputs.

Rules: Are the rules defined by the control developer. These rules are stated as condition-action rules. There are three types of rules/specifications:

Global Rules: Inter-agent requirements which can be changed at the system configuration level.

Local Rules: Intra-agent requirements changeable at the agent configuration level. Local rules are divided into three categories, namely "Pre-conditions," "Post-actions" and "Post-states." *Post-actions* are functions that must be executed by an agent after an action is taken. *Post-state* or *Transition function* δ is a mapping from $S \times F$ to S. Transition functions change the state of the agent upon execution of a basic function.

Layout Specifications: Inter-agent requirements which define the accessibility between agents, that is, which other agents are accessible (physically or logically) by an agent.

Flags: A low/high signal triggered by the agent when the execution of a job is finished.

Attributes: Are the characteristics of the agents.

Initiators: Are flags that initiate a sequence of tasks to be executed by an agent. Each initiator is associated with one or more ordered goal functions. Initiators will be activated by request messages that the agent receives from its requesters.

2.2 Architecture of an agent

As shown in Fig. 1 an agent is composed of a "core," a "communication layer," a "pre-execution layer," an "execution layer" and a "memory "The core has a component called the *synthesizer*. The synthesizer is responsible for finding the task schedule of the agent. It also triggers algorithms related to fault detection and recovery. The core is in continual connection with the communication layer. Due to the distributed nature of the system, the communication layer is responsible for setting up the messages that have to be sent to the other agents, requesters or providers alike. Receiving information on the state of other agents is also done through communication unit. Different outputs of the core have to be transferred to the pre-execution and communication layers. The model synthesized by the agent core is executed by the agent's execution layer. The collected local and global information can be stored by the agent in its memory unit.

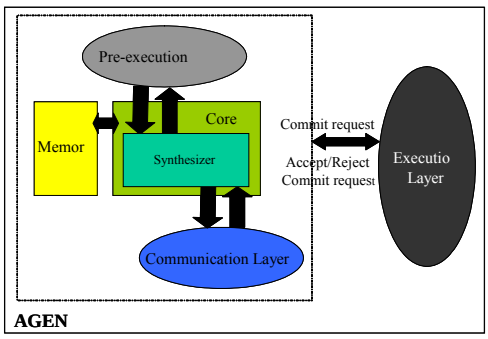

Figure 1: Agent architecture

We comply with the specifications in the FIPA (Foundation for Physical Intelligent Agents). Each agent has an identifier and this must be registered by another agent called the *bookkeeper agent*. In FIPA this component is called Agent Management System (AMS). The communication layer in our model is equivalent to Message Transport Service (MTS) in FIPA. The difference is that in FIPA each Agent Platform (AP) has its own individual MTS but in our model every agent has a separate MTS.

In this framework the communication between agents is handled based on a requester/provider protocol. Agents issue request messages and send them to all of their existing providers and wait for the best proposal. The providers compete to get the jobs by bidding. Whichever agent has the best offer wins the corresponding job and will be assigned by the requester as the provider of the job and the other agents will update their flags thereafter. Fig. 2 shows part of the sequence diagram of the communication process. Different scenarios can occur during communication process, depending on the state of the plant and agents. Discussion about the details of the sequence diagram of the communication is outside of the scope of this paper.

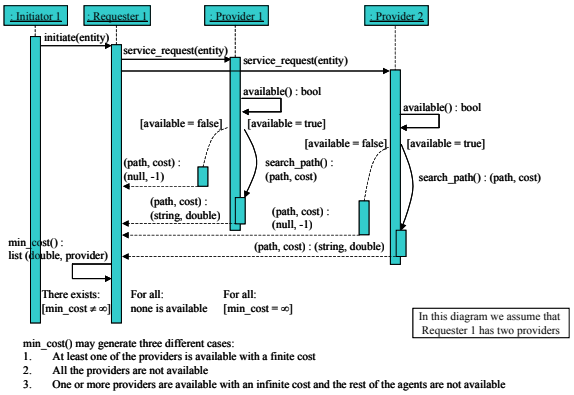

Figure 2: Part of the sequence diagram of the messaging protocol

Agents will initiate their synthesizers upon reception of a request. Their synthesizers search for a road map (controlled solution) to execute the requested job in an appropriate way. This includes not only the satisfaction of the local and global rules but also a solution that minimizes the cost of the execution. If such a path is found by the synthesizer, the provider agent announces its price to the requester. The bidding policy in this paper is first-price sealed bid auction. As soon as a provider agent wins a bid, the job will be added to the work-to-do list of the agent (part of its memory unit). The pre-execution layer will be activated when the agent wants to execute the job physically. In this layer another synthesis and scheduling process will be performed, because the condition of the agent and the environment may have been changed during the time of bidding and

execution. The output of the pre-execution layer will be fed to the execution layer.

Commit Stage. As soon as the requester agent selects its provider, a "commit request" is created and sent to the selected agents. These agents will then set their state to a "locked" condition.

2.3 Synthesis

In its simplest form, the provider's synthesizer uses a search algorithm to obtain a feasible solution path (using a depth-first search method) from its current state to the desired goal state based on its local specifications and basic functions. During the path generation the algorithm also calculates the cost of the generated path according to the specified cost function.

Initiator of an agent defines the goal function(s) that must be searched from the current state of the agent. The algorithm initially starts at the root of the search tree. The number of the total possible branches of the root is equal to the total number of the basic functions that the agent possesses. Each branch is equivalent to the execution of the corresponding basic function. If all the pre-conditions of a specific basic function are satisfied, the algorithm generates a new node and goes to the next level of the tree. If this is not the case that branch will not be part of the solution and the tree will have a dead end at that point and the algorithm will continue with the search at a previous level of the tree. If a basic function is considered as executed (new node generated), then the subsequent steps will be to update the states (based on the transition functions) and then to execute all the post-conditions, which means the generation of a new node. This procedure continues until a solution has been found or not found. The reader is referred to (Amini et al., 2005) for more details. Using this method we can easily reconfigure agents by changing their rules. No new control design will be necessary.

Deadlock detection and prevention. It is possible that all providers respond with an infinite cost to a request from a requester. Two cases are possible: The requester waits for certain time until one of the providers becomes available and offers its service. The second case is when deadlock condition happens. A deadlock occurs, because two or more agents require the same resources during a particular time period in some circular-wait manner. Hence the providers can never find any solution path to reach to their desired goals. In these situations we have to detect the deadlock state and then prevent the system to enter to this state in the future. Fanti and Zhou (2004) and Shih and Stankovic (1990) give a comprehensive description of deadlock detection/prevention/avoidance in manufacturing automation and computer science community, respectively.

In a distributed environment, there cannot be a centralized control that usually manages deadlocks. Therefore, there must be a framework through which agents can communicate and solve the problem of deadlock amongst themselves. This involves three steps: (1) Determine those resources that cause the provider agent not to find any solution path. (2) Detect deadlocks. (3) Reconfigure the system so that the future deadlocks of the same type and origin are avoided. Fig. 3 illustrates the concept.

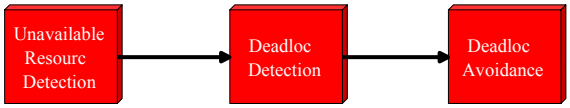

Figure 3: Deadlock detection and avoidance steps

We have developed two algorithms: a resource detection algorithm to detect unavailable resources, and a deadlock detection algorithm using Mitchell and Merritt algorithm (1984) to detect deadlocks. Should a deadlock be detected, the synthesizer triggers another algorithm, the deadlock *avoidance algorithm,* to avoid the same situation in the future. For more details, the reader is referred to (Amini et al., 2005).

3. ILLUSTRATIVE EXAMPLE

Consider the system shown in Fig. 4, which consists of one input buffer, one output buffer, two robots and three machines and one part type. Buffer I_1, robots R_1 and R_2, and machines M_1, M_2, M_3 define our agents in this system. Parts (of single type) are the entities which follow a process plan (global specification). Since there is no supervisory control, every single agent must work according to its own specifications.

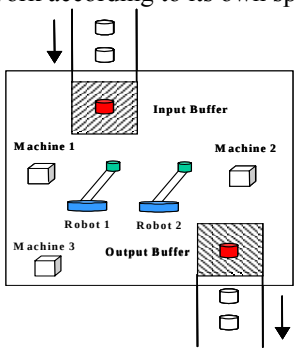

Figure 4: Illustrative Flexible Manufacturing Cell

Due to the lack of space we only present the specifications of the robots.

Robot $\Phi = R_i$ (i = 1, 2)

Description: R_1 and R_2 are responsible for moving parts between different stations. We assume that the robots can hold at most one part in their hands. In our notation E is the entity (part) that has to be processed.

Attributes ($R_i.A$): $R_i.A.a_1$(capacity) = 1,
$R_1.A.a_2$(reachability) = {I_1, M_1, M_2, M_3, O_1}
$R_2.A.a_3$(reachability) = {M_2, M_3, O_1}

States ($R_i.S$): (s_1, s_2, s_3, s_4, s_5) = (*stage*, *occupancy*, *location*, *current_part_id_in_process)*: (idle/busy, empty/not empty, $I_i/O_i/M_i$, E)

Basic Functions ($R_i.F$):
$R_1.F.f_1$ = move_to(Y, E), where $Y \in$ {M_1, M_2, M_3, I_1, O_1}
$R_1.F.f_2$ = take(Y, E) where $Y \in$ {M_1, M_2, M_3, I_1}$R_1.F.f_3$ = put(Y, E) where $Y \in$ {M_1, M_2, M_3, O_1}
$R_2.F.f_1$ = move_to(Y, E), where $Y \in$ {M_2, M_3, O_1}
$R_2.F.f_2$ = take(Y, E) where $Y \in$ {M_2, M_3}
$R_2.F.f_3$ = put(Y, E) where $Y \in$ {M_1, M_2, O_1}

Flags ($R_i.f$): None
Initiator ($R_1.In$):
$R_1.In.I_1.\varphi_1 = 1$ or $R_1.In.M_i.\varphi_2 = 1$: take(X_i, E) $\rightarrow$ put($E.PP_i(k+1)$, E), where $X_i \in$ {M_1, M_2, M_3, I_1},
($R_2.In$):$R_2.In.M_i.\varphi_2 = 1$:take(X_i,E) $\rightarrow$ put($E.PP_i(k+1)$, E), where $X_i \in$ {M_2, M_3}

Rules ($R_i.R$):

- $R_1.R.r_1$: rules for move_to(Y, E):
  Pre-conditions:
  $Y \in R_1.A.a_2$
  $R_1.S.s_3 \neq Y$
  Post-actions: None
  Post-states:
  $R_1.S.s_3 = Y$
  $R_1.S.s_4 = E$
- $R_1.R.r_2$: rules for take(Y, E):
  Pre-conditions:
  $Y \in R_1.A.a_2$
  $R_1.S.s_1$ = -
  $R_1.S.s_2$ = empty
  $R_1.S.s_3 = Y$
  Y.out = true
  Post-actions:
  move_to(W, E) where W is the next station in the processing sequence after Y and E = (PP_i, j, k + 1)
  Post-states:
  $R_1.S.s_2$ = not empty
  $R_1.S.s_3 = Y$
  $R_1.S.s_4 = E$
- $R_1.R.r_3$: rules for put(Y, E):
  Pre-conditions:
  $Y \in R_1.A.a_2$
  $R_i.S.s_1$ = idle
  $R_i.S.s_2$ = not empty
  $R_i.S.s_3 = Y$
  $R_i.S.s_4 \neq$ Null
  Y.in = true
  Post-actions: None
  Post-states:
  $R_1.S.s_2$ = empty
  $R_1.S.s_3 = Y$
  $R_1.S.s_4 = E$

Similar specifications hold for R_2 and other agents.

Agent R_1 is accessible by I_1, M_1, M_2, M_3, O_1 and R_2 by M_2, M_3 and O_1. The state of the robot is determined by a 5-tuple: The first value can be either idle (when there is no part in its hand and does not move) or busy (if it has a part in its hand or is moving towards a machine or buffer). The second value defines robot occupancy. It will be either empty or not empty. The next value is the robot's location. Since the robot is moving around in the cell, this value has to be tracked. The state of the part currently held by the robot is the next element in the tuple (this value will be NULL if there is no part carried by the robot).

R_1 has three basic functions available. Using *move_to*(Y, E) function it will move from its current location towards location Y, to perform a task related to entity E. Function *take*(Y, E) will be responsible for taking part E form agent Y. And *put*(Y, E) function is responsible for putting part E into agent Y.

The process plan is defined by PP_1={I_1, M_1, M_2, M_3, O_1}. Consider the state of R_1 to be $R_1.S$.(idle, empty, M_1, NULL), and the state of the M_1 to be $M_1.S$.(idle, 0, NULL) (0 is number of parts currently being processed in the machine). Now suppose that part E(PP_1, 1, 1) arrives in input buffer I_1. PP_1 means that the arriving part is of type 1. The first "1" in the tuple means that this part is the first instance of this entity type arriving into the cell. The second "1" means that the part is currently in its first station (I_1). At this time $I_1.f.\varphi_1$ becomes equal to one. I_1 then sends a request message to R_1, since R_1 is the only provider of I_1. R_1 will receive the request message that part E is ready to be picked up. From its list of specifications, R_1 will find the required target(s). According to the specifications ($R_1.In.I_1.\varphi_1 = 1$), the objective of R_1 will be to take(I_1, E)$\rightarrow$put($E.PP_1$(2), E). Which means, to take the part from the input buffer and put it in M_1 (The second station in PP_1 is M_1). At this point the robot will start to construct its search tree in the synthesis process. Since the location of R_1 is currently M_1, it cannot take the part from the input buffer (a pre-condition for the basic function take(I_1, E) is $R_1.S.s_3 = I_1$, which is currently not true). The only possible action that the robot can

take is move_to(I_1, E), since all the pre-conditions for this action are satisfied. If the robot performs it, its next state will become $R_1.S.$(busy, empty, I_1, NULL) based on the transfer functions (post-states). Since the pre-conditions for take(I_1, E) are satisfied now, the agent can perform this function next and therefore the resulting state of the agent will become $R_1.S.$(busy, not empty, I_1, $E(PP_1, 1, 1)$). Given that *take* function has a post-action, the robot will do the post-action first. This action is going to be move_to(M_1, $E(PP_1, 1, 2)$). Since all the pre-conditions are satisfied, it will perform this step and therefore the next state of R_1 will become $R_1.S.$(busy, not empty, M_1, $E(PP_1, 1, 2)$). Now the only basic function that can be executed by the robot is *put*. One of the pre-conditions of put function is Y.in = true, where Y is M_1. *in* and *out* functions are related to the communication and are used to get the state of the other agents in this distributed environment. The state of M_1 is currently $M_1.S.$(idle, 0, NULL). The robot calls the public function (*in*) of the machine. If all the pre-conditions of M_1 for this function are satisfied (i.e. if the machine can accept a new part as input), it will return "true" and therefore the robot can put the part into the machine and by doing this, the request can be executed. At the execution stage R_1 will "lock" all the necessary resources (I_1 and M_1) and will commit this job henceforth.

After M_1 finishes its job on the part it will announce this event to its providers (only R_1). Since R_1 is at idle state, it will be able to take the part from M_1 and put it into M_2. M_2 finishes its job and sends a request to both of its providers, namely R_1 and R_2 with current states of them being $R_1.S.$(idle, empty, M_2, NULL) and $R_2.S.$(idle, empty, M_3, NULL), respectively. The two robots receive a message from M_2 to move the outstanding part to M_3, and both are able to find a solution. But since R_1 is closer to the M_2, its cost is estimated at a lower value and therefore it will offer a better price. Hence M_2 will select R_1 as its provider. The last step from M_3 to O_1 will be executed by R_2 since it is the only robot connected to O_1.

4. IMPLEMENTATION

We have used **IEC 61499** standard for the implementation of the above distributed control systems. This standard is based on "function blocks" and it provides an easily distributable methodology combined with an event-driven data exchange. For more details the reader can refer to (Amini et al., 2005).

5. CONCLUSION

We have developed a framework for control synthesis of distributed agent system. Each agent has its own controller, which can be reconfigured by changing its local specifications and/or the global specifications. The synthesizer is capable of detecting and avoiding single level deadlocks at execution level.

REFERENCES

Amini, A., P. Zhao, and M.A. Jafari (2005). Control synthesis for distributed multi agent systems. Manuscript under revision.

Fanti, M.P. and M. Zhou (2004). Deadlock Control Methods in Automated Manufacturing Systems. *IEEE Transactions on Systems, Man, and Cybernetics—Part A: Systems and Humans*, **vol. 34**, no. 1.

Hanisch, H-M. and M. Rausch (1995). Synthesis of Supervisory Controllers based on Novel Representation of Condition/Event Systems. *Proceedings of IEEE International Conference on Systems, Man and Cybernetics*, **vol. 4**., pp. 3069-3074.

Krogh, B.H. and S. Kowalewski (1996). State Feedback Control of Condition/Event Systems. *Mathematical and Computer Modelling*, **vol. 23**, no. 11/12, pp. 161-173.

Mitchell, D.P. and M.J. Merritt (1984). A Distributed Algorithm for Deadlock Detection and Resolution. *Proceedings of the Third Annual ACM Symposium on Principles of Distributed Computing*, *ACM SIGACT SIGOPS*, pp. 282-284.

Ostroff, J.S. (1989). Synthesis of controllers for real-time discrete event systems. *IEEE Proceedings of the 28th Conference on Decision and Control*, Tampa, Florida, pp. 138-144.

Ramadge, P.J. and W.M. Wonham (1987a). Supervisory Control of a Class of Discrete Event Processes. *SIAM Journal on Control and Optimization,* **vol. 25**, no.1, pp. 206-230.

Ramadge, P.J. and W.M. Wonham (1987b). On the Supremal Controllable Sublanguage of a Given Language. *SIAM Journal on Control and Optimization*, **vol. 25**, no. 3, pp. 637-659.

Shih, C.-S. and J. Stankovic (1990). Survey of Deadlock Detection in Distributed Concurrent Programming Environments and Its Application to Real-Time Systems and Ada. *Technical Report UM-CS-1990-069*, University of Massachusetts, Amherst, MA.

Zhou, M.C. and F. DiCesare (1993). *Petri Net Synthesis for Discrete Event Control of Manufacturing Systems*. Kluwer Academic Publishers, Netherlands.