



# Diagnosability for Patterns in Distributed Discrete Event Systems

Yuhong Yan<sup>1</sup>, Lina Ye<sup>2</sup>, and Philippe Dague<sup>2</sup>

<sup>1</sup> CSE, Concordia University, Montreal, Canada  
yuhong@cse.concordia.ca

<sup>2</sup> LRI, Univ. Paris-Sud 11, France  
{lina.ye, philippe.dague}@lri.fr

## ABSTRACT

A pattern is a Finite State Machine that can describe rich faulty scenarios, such as the occurrence of single faults, multiple faults, multiple occurrences of a fault, or the repair of a system. In distributed systems, the events in the pattern, as well as in the system trajectories, are emitted from different components. Our approach is based on distributed simulation and communication to check the recognition of the pattern from the conclusion of local recognition of local patterns. The components communicate observable events and shared communication events, as well as their local recognition results during the checking process without sharing their local models in any way.

## 1 INTRODUCTION

Discrete Event Systems (DESs) are widely used to describe system dynamics. Classically, a fault is associated with one non-observable faulty event. The fault diagnosis task in discrete event systems is to reason about the occurrence of a fault from a sequence of observed events emitted. If a finite sequence of observable events can non-ambiguously identify a fault from the other faulty or correct behaviors, this fault is called diagnosable.

A pattern is modeled as a Finite State Machine (FSM) with the events from the original system (Jéron *et al.*, 2006). It can uniformly describe various faulty scenarios, such as single fault, multiple faults, multiple occurrences of a fault, and the repair of a system after the occurrence of a fault, etc. For example, a single fault can be modeled as a FSM that includes one transition that emits a faulty event; and a pattern for two faults can be modeled as an FSM that contains two branches from the initial state and each of the branches presents a scenario that one of the two faulty events precedes another respectively (Jéron *et al.*, 2006). Indeed, a pattern is a unified description of rich faulty scenarios. Thus, studying pattern diagnosis and diagnosability is of great interest. A system can be diagnosed by deducing the recognition of the pattern from

a sequence of observed events. The purpose of pattern diagnosability is to answer whether this recognition is deterministic for all the trajectories with the same observations.

Our paper is to solve the pattern diagnosability problem in distributed discrete event systems. Our approach is based on distributed simulation, which is new in the research of distributed diagnosability analysis. We align local trajectories into partially observable global trajectories by distributed simulation. The global consistency is ensured via the simulation, instead of calculating a verifier (Pencolé, 2004). Further, we develop a process to compute globally observed traces and recognition of the pattern, and further compute the pattern diagnosability from the local recognition results. We have proved the correctness of our method. One advantage of our approach is that we avoid expensive operation to calculate local diagnosers (Pencolé, 2004; Ye *et al.*, 2009) and verifier (Pencolé, 2004). Another advantage of our approach is that the components do not share their local models in any direct or indirect way (e.g. sharing their local diagnosers (Pencolé, 2004)).

## 2 PRELIMINARIES

A distributed DES is composed of a finite set of components,  $\{G_0, G_1, \dots, G_n\}$ , each of which is modeled as an FSM (Definition 1).

**Definition 1 (Local Model).** *The local model of a component  $G_i$  is an FSM  $G_i = (Q_i, \Sigma_i, \delta_i, q_i^0)$ , where*

- $Q_i$  is a finite set of states
- $\Sigma_i$  is a finite set of events
- $\delta_i \subseteq Q_i \times \Sigma_i \times Q_i$  is a set of transitions
- $q_i^0$  is the initial state

The set of events  $\Sigma_i$  is divided into three parts  $\Sigma_i = \Sigma_{i_o} \uplus \Sigma_{i_u} \uplus \Sigma_{i_c}$ , where  $\Sigma_{i_o}$  is the set of locally observable events which are disclosed to any other components;  $\Sigma_{i_u}$  is the set of locally unobservable events; and  $\Sigma_{i_c}$  is a set of communication events which are shared by  $G_i$  and at least one of the other components, and are observable only to their owners. As the locally observable events are disclosed to

the other components, they can be considered globally observable as well in this paper. We assume that an event can be labeled its owner's name. Therefore, for any pair of local components  $G_i$  and  $G_j$ , we have  $\Sigma_{i_o} \cap \Sigma_{j_o} = \emptyset$  and  $\Sigma_{i_u} \cap \Sigma_{j_u} = \emptyset$ . Globally, we have  $\Sigma_o = \bigsqcup_{i=1}^n \Sigma_{i_o}$ ,  $\Sigma_u = \bigsqcup_{i=1}^n \Sigma_{i_u}$  and  $\Sigma_c = \bigcup_{i=1}^n \Sigma_{i_c}$ . For convenient,  $\delta_i \subseteq Q_i \times \Sigma_i \times Q_i$  is also used to represent  $\delta_i \subseteq Q_i \times \Sigma_i^* \times Q_i$  in the following way:

- $(q, \epsilon, q) \in \delta_i$ , where  $\epsilon$  is the null event
- $(q, se, q_1) \in \delta_i$  if  $\exists q' \in Q, (q, s, q') \in \delta_i$  and  $(q', e, q_1) \in \delta_i$ , where  $s \in \Sigma_i^*, e \in \Sigma_i$

We assume that the components do not share their local model with the other components and there is no central unit that knows all the local models. We further assume that the components do not synchronize their local clocks. That means the observed sequence of observable events from different components is not necessarily the same as their occurrence sequence. If one event from one component occurs before a communication event and another event from a second component occurs after the communication event, we can determine their sequences. Otherwise, we deduce the two events can occur concurrently.

Because the components are assumed to work continuously, the models of the components are **alive**, i.e.  $\forall q \in Q_i, \exists e \in \Sigma, (q, e, q') \in \delta_i$ . The prefix-closed language  $L(G)$  generated by  $G$  describes the behavior of the system, which implies all the states in  $G$  are final states. We will use **trajectories** instead of words:

$$L(G) = \{s \in \Sigma^* \mid \exists q \in Q, (q^0, s, q) \in \delta\}$$

In our paper, both the generated language and the recognized language are assumed to be **observable alive**, which means that there is no cycle containing only unobservable events. Given a trajectory  $s \in L$ , we denote  $L/s$  as the post-language of  $L$  after  $s$  and denote  $P_{\Sigma_o}(s)$  as the sequence of observable events in  $s$ .  $P_{\Sigma_o}(s)$  ( $P(s)$  when no ambiguity) is also called a **trace**. The inverse projection of a trajectory  $s$  is denoted as  $P^{-1}(P(s)) = \{t \in L(G) \mid P(t) = P(s), s \in L(G)\}$ .

**Definition 2 (Synchronization).** Given two FSMs  $G_1 = (Q_1, \Sigma_1, \delta_1, q_1^0)$  and  $G_2 = (Q_2, \Sigma_2, \delta_2, q_2^0)$ , their synchronized product based on the communication events is  $G_{1\parallel 2} = G_1 \parallel_{\Sigma_{1_c} \cap \Sigma_{2_c}} G_2 = (Q_1 \times Q_2, \Sigma_1 \cup \Sigma_2, \delta_{1\parallel 2}, (q_1^0, q_2^0))$ , where  $\delta_{1\parallel 2}$  is defined in the following:

- $\{((q_1, q_2), \sigma, (q_1', q_2'))\}$ , if  $\sigma \in \Sigma_{1_c} \cap \Sigma_{2_c}$  and  $(q_1, \sigma, q_1') \in \delta_1, (q_2, \sigma, q_2') \in \delta_2$ .
- $\{((q_1, q_2), \sigma, (q_1', q_2))\}$  if  $\sigma \in \Sigma_1 \wedge \sigma \notin \Sigma_2$  and  $(q_1, \sigma, q_1') \in \delta_1$ .
- $\{((q_1, q_2), \sigma, (q_1, q_2'))\}$  if  $\sigma \notin \Sigma_1 \wedge \sigma \in \Sigma_2$  and  $(q_2, \sigma, q_2') \in \delta_2$ .
- otherwise  $\delta_{1\parallel 2}$  is undefined.

Figure 1 depicts a distributed system with components  $\{G_1, G_2\}$ . For  $G_1$ ,  $\Sigma_1 = \{c1, c2, u1, u2, o1\}$ ,  $\Sigma_{1_o} = \{o1\}$ ,  $\Sigma_{1_u} = \{u1, u2\}$ ,  $\Sigma_{1_c} = \{c1, c2\}$ .  $G_2$  has similar definitions. The global model of the system is  $G = \{Q, \Sigma, \delta, q^0\} = G_1 \parallel G_2$ . We try to avoid to use this operation to get the global model due to

its computational expensiveness and non-sharable local models.

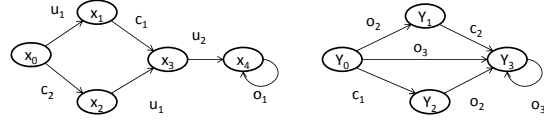


Figure 1: A distributed system with two components  $G_1$ (left) and  $G_2$ (right).

### 3 PATTERN DIAGNOSABILITY

From domain knowledge, we can associate a pattern with the occurrence of single or multiple faults, or the repair of a system (Jéron *et al.*, 2006). A pattern can be described as an FSM (Definition 3).

**Definition 3 (Pattern).** A pattern is an FSM with final states set  $F_\Omega$ ,  $\Omega = (Q_\Omega, \Sigma_\Omega, \delta_\Omega, q_\Omega^0, F_\Omega)$ , which satisfies the following conditions:

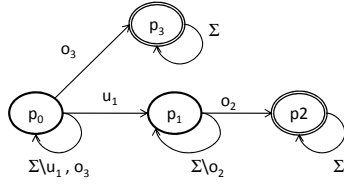
- $\forall q \in Q_\Omega, \forall \sigma \in \Sigma_\Omega$ , if  $(q, \sigma, q_1) \in \delta_\Omega$  and  $(q, \sigma, q_2) \in \delta_\Omega$ , then  $q_1 = q_2$ .
- $\forall q \in Q_\Omega, \Sigma_\Omega(q) = \Sigma_\Omega$  where  $\Sigma_\Omega(q) = \{\sigma \in \Sigma_\Omega \mid \exists q' \in Q_\Omega, (q, \sigma, q') \in \delta_\Omega\}$ .
- $F_\Omega \subseteq Q_\Omega$  and  $\delta_\Omega(F_\Omega, \Sigma_\Omega) \subseteq F_\Omega$  where  $\delta_\Omega(F_\Omega, \Sigma_\Omega) = \bigcup_{q \in F_\Omega, \sigma \in \Sigma_\Omega} \{q' \in Q_\Omega \mid (q, \sigma, q') \in \delta_\Omega\}$ .
- $q_\Omega^0 \notin F_\Omega$

The first two conditions describe the pattern as a deterministic and complete finite state automaton. The third condition characterizes that the final state set  $F_\Omega$  is stable. The set of all the trajectories that transfer the FSM to its final states is a recognized language of this FSM. Then it can be deduced that its recognized language  $L_m(\Omega)$  is "extension-closed", formally described as

$$\forall s \in L_m(\Omega), \forall st \in \Sigma_\Omega^*, sst \in L(\Omega) \rightarrow sst \in L_m(\Omega)$$

which means that once the system arrives in a final state, it will be always in a final state in the future. As  $\Omega$  is complete,  $ssst \in L(\Omega)$  is always true. Given a system FSM  $G$  and a pattern  $\Omega$ , we assume:  $\Sigma_\Omega = \Sigma$ ,  $\Sigma_{\Omega_o} = \Sigma_o$ . Notice that the FSM for pattern has final states to mark the recognition of a fault, while the models for components do not have final states. Figure 2 presents a pattern  $\Omega$  for the system displayed in Figure 1.

A system can be diagnosed by deducing the recognition of the pattern from a sequence of observed events. Pattern diagnosability is to answer whether this recognition is deterministic for all the trajectories with the same observations. In addition, we want the recognition process to be bound, i.e. there must exist  $n \in \mathbb{N}$ , where  $\mathbb{N}$  is the set of nature numbers, that whenever  $s \in L(G) \cap L_m(\Omega) \cap \Sigma_o^*$ , for all  $t \in L(G)/s \cap \Sigma_o^*$ , the inverse projection of  $P(st)$  is also recognized by  $L_m(\Omega)$ . The term  $\Sigma_o^*$  constrains the trajectory to end with an observable event

Figure 2: Pattern  $\Omega$ .

for the convenience of recognition. The following definition follows the bound diagnosability in (Jéron *et al.*, 2006):

**Definition 4 (Pattern Diagnosability).** A system FSM  $G$  is  $\Omega$ -diagnosable if  $\exists n \in \mathbb{N}$ ,  $\forall s \in L(G) \cap L_m(\Omega) \cap \Sigma^* \Sigma_o$ ,  $\forall t \in L(G)/s \cap \Sigma^* \Sigma_o$ , if  $|t| \geq n$ , then  $P^{-1}P(st) \subseteq L_m(\Omega)$ .

Definition 4 says that if exists a nature number  $n$ , for any trajectory  $s$  satisfying  $s \in L(G) \cap L_m(\Omega) \cap \Sigma^* \Sigma_o$ , and any sequence of events  $t$  satisfying  $t \in L(G)/s \cap \Sigma^* \Sigma_o$ , if  $t$  is longer than  $n$ , then the inverse projection of the projection of  $st$  is recognized by the pattern  $\Omega$ . In another word, all the possible trajectories of what we can observe (which is the projection of  $st$ ) is recognized by the pattern  $\Omega$ , if we wait long enough. This means we can non-ambiguously decide that the pattern is recognized from what we have observed, if we wait long enough. This is the exact meaning of  $\Omega$ -diagnosability.

## 4 PATTERN DIAGNOSABILITY IN DISTRIBUTED DISCRETE EVENT SYSTEMS

### 4.1 The Principle

If we want to prove a system is  $\Omega$ -diagnosable, we need to follow Definition 4. If we want to prove a system is not  $\Omega$ -diagnosable, we can follow Theorem 1.

**Theorem 1** *If there exist two infinite trajectories for a system which have the same trace, one of which is recognized by the pattern  $\Omega$  and the other is not, the system is not  $\Omega$ -diagnosable.*

**Proof:** We use reductio ad absurdum. Assume two infinite trajectories  $s_1, s_2 \in L(G) \cap \Sigma^* \Sigma_o$  have the same trace, i.e.  $P(s_1) = P(s_2)$ . And assume  $s_1$  is recognized by  $\Omega$ , while  $s_2$  is not, i.e.  $s_1 \in L_m(\Omega)$  and  $s_2 \notin L_m(\Omega)$ .

Assume  $t$  is any infinite sequence of events after  $s_1$  is recognized by the pattern, i.e.  $\forall t, s_1 = st \wedge s \subseteq L_m(G)$ . Meanwhile we have  $s_2 \in P^{-1}P(st)$  and  $s_2 \notin L_m(\Omega)$ . Therefore, there is no such a number  $n$  that can satisfy Definition 4, i.e.  $\nexists n$ , such that if  $|t| \geq n$ , then  $P^{-1}P(st) \subseteq L_m(\Omega)$ .  $\square$

Based on Theorem 1, if we want to prove a system is not  $\Omega$ -diagnosable, we need to find out at least two infinite trajectories with the same trace, one of which is recognizable by the pattern, and the other is not. If we want to prove a system is  $\Omega$ -diagnosable, we verify all the infinite trajectories such that if their traces are the same, they should be uniformly recognized or not recognized by the pattern.

In a distributed system, the principle is the same. The difficulty is that we cannot directly obtain the trajectories, as the components do not share their models. We use distributed simulation to identify globally consistent trajectories, and then determine if the trajectories with the same trace are uniformly recognized by the pattern or not. The two definitions below are used in the rest of the paper.

**Definition 5** *The critical transitions of a pattern are the transitions in the pattern whose starting and ending states are not the same. And the critical events are the events emitted by the critical transitions.*

The critical transitions in Figure 2 are  $(p_0, u_1, p_1)$ ,  $(p_0, o_3, p_3)$ , and  $(p_1, o_2, p_2)$ . The critical transitions are critical because they progress the process of recognizing a pattern. The critical events are  $\{u_1, o_2, o_3\}$ .

**Definition 6** *A local pattern  $\Omega_i$  for a component  $G_i$  is an FSM modified from  $\Omega$  by keeping only the critical transitions and renaming the local non-observable events which do not belong to  $G_i$  to a silent event  $\epsilon$ .*

For  $\Omega$  in Figure 2, the local pattern  $\Omega_1$  for component  $G_1$  is the same as  $\Omega$  except the looped transitions are removed (see also Fig. 3 left). The local pattern  $\Omega_2$  for component  $G_2$  renames  $u_1$  to  $\epsilon$  and it is without the looped transitions (see also Fig. 3 right).

Figure 3: The local patterns for  $G_1$  (left) and  $G_2$  (right).

### 4.2 Distributed Simulation

We assume that the components can simulate their local models and can send and receive messages to communicate their local observable events and communication events with one another. Via simulation, the components identify whether their local patterns are recognized, and then exchange the local recognition results. Each component can conclude the same result about the global diagnosability by synthesizing the local recognition results it receives. This diagnosis analysis is conducted off line, i.e. no observations at the run time are used. We first present some data structures and then the simulation algorithm.

When a component simulates its model, it starts from the initial state and explores all the possible branches at the same time. An **execution tree** is a tree such that its root node is the initial state and its branches are the transitions traversed by a breadth first search algorithm on the model. Since the local model is alive, an execution tree can grow infinitely. An execution path (Definition 7) is a path from the root node of an execution tree to one of its leaf nodes.

**Definition 7** *An execution path for a local model  $G_i = (Q_i, \Sigma_i, \delta_i, q_i^0)$  is an alternating sequence of states and events ending with a state:  $\rho = q_i^0 \alpha_1 q_i^1 \alpha_2^2 \dots q_i^n$ , such that  $q_i^j \in Q_i$ ,  $\alpha_i^{j+1} \in \Sigma_i$  and  $(q_i^j, \alpha_i^{j+1}, q_i^{j+1}) \in \delta_i$ .*

The execution path and the execution tree represent the actual execution sequence during simulation. Since the states are finite, the simulation will revisit a state in finite steps. Therefore, though the execution path is infinite, we can define a finite execution graph (Definition 8) to represent the simulation process. The execution graph grows similarly as the execution tree, except that a transition ending with a visited state creates a loop, while in the execution tree, a path grows infinitely long.

**Definition 8** An *Execution Graph* for an FSM  $G_i = (Q_i, \Sigma_i, \delta_i, q_i^0)$  is also an FSM  $G_E = (Q_E, \Sigma_E, \delta_E, q_E^0)$  built by breadth first searching  $G_i$ :

- $q_i^0 \mapsto q_E^0$ .
- Assume a state  $q_i^1 \in Q_i$  has  $n$  outgoing transitions, i.e.  $\{(q_i^1, e, q_i^j) \in \delta_i, j \in [1, n]\}$ . Each of the transitions should have a correspondent transition in  $G_E$ , i.e. if  $q_i^1 \mapsto q_E^1$ , then any  $q_i^j \mapsto q_E^j$ , and  $(q_i^1, e, q_i^j) \mapsto (q_E^1, \{e, id\}, q_E^j)$ , where  $id$  is a unique  $id$  defined latter.
  - If any of the end states in  $\{q_i^j\}$  is visited before, its correspondent state should remain the same in  $G_E$ .
  - If  $q_i^j = q_i^{j'}$ , where  $j, j' \in [1, n]$ , are two end states which are the same and not visited before, their correspondent states in  $G_E$  should be different, i.e.  $q_E^j \neq q_E^{j'}$ .

The  $id$  is a unique label for a transition in the execution graph. The  $id$  provides a convenient way to trace the execution paths between the components (cf. the next subsection). The  $id$  is  $id_1 \in N$  at the first level and in the format of  $id_{n-1}.k$  ( $k \in N$ ) at the  $n$ -th level, where  $N$  is the set of nature numbers. With an  $id$ , we can easily retrieve the execution path that leads to the current transition. A method  $id = T.grows(t)$  grows the tree  $T$  with the given transition  $t$  and returns the generated  $id$ . A method  $T.getPath(id)$  returns an array of transitions on the execution path that leads to the transition labeled  $id$ .  $G_i.id$  is to tell the  $id$  belongs to a component  $G_i$ .

**Example 1** The execution graphs of  $G_1$  and  $G_2$  in Figure 1 are shown in Figure 4. *{By Yuhong: here can add another example}*.

Since the execution graph is finite, it is possible that we can stop simulation after traversing all the transitions in the execution graph. The following theorem is important to guarantee the algorithm developed in this paper terminates.

**Theorem 2** It takes finite steps to traverse the execution graph.

Proof: as the execution graph is an FSM, the conclusion is obvious.

If a system has only one component, we can stop simulation if we have traversed all the transitions in the execution graph. For a distributed system, fundamentally, since the whole system model can be obtained by synchronizing the local models, it is also an FSM. Thus, we just need to simulate finite steps before

the whole system goes into loops. However, since we do not synchronize the local models, we can indirectly know if the whole system goes into a loop by checking if each of the components repeats the same transitions at one point (cf. below).

**Communications and messages.** A message  $m$  has a message type  $m.type \in \{\text{“communicate”}, \text{“confirm”}, \text{“loop”}\}$ . The message content for “communicate” and “confirm” messages is a tuple  $\langle e, G_i.id, G_j.id \rangle$ , in which  $e$  is an event,  $G_i$  is the sender component,  $G_j$  is the receiver component. The message content for “loop” messages is  $\langle G_i.id1, G_i.id2, G_j.id \rangle$ , where the path between  $G_i.id1$  and  $G_i.id2$  is a loop. If any fields in the messages are unknown,  $null$  is used.  $m = sends(type, \langle e, G_i.id, G_j.id \rangle)$  and  $m = sends(type, \langle G_i.id1, G_i.id2, G_j.id \rangle)$  compose the message  $m$  and send it to  $G_j$ .  $G_j.id$  is the last transition  $id$  in the matched trajectory in  $G_j$  (cf. below). If no matching trajectory exists,  $G_j$  is used. The communication protocol is as following:

1. When a communication event is emitted, the component sends a “communicate” message to all components which share this communication events (possibly more than one), and waits for “confirm” messages from all its correspondents before resuming the execution on the branch. “Time-out” can remove any unsynchronizable path from the execution paths pool. A method  $b = m.matches(m'.e, id)$  matches a message  $m$  with a message  $m'$ . It returns 1 iff  $m.e == m'.e \wedge m.G_j.id \geq id$ , otherwise  $b = 0$ .  $m.G_j.id \geq id$  holds if the current  $id$  in the receiver (aka  $G_j$ ) is a subbranch of  $G_j.id$ .  $G_j$  is considered as a root. A path in a component can match multiple paths in another component. To reuse the already built paths, we keep all the confirmed messages in memory.
2. When an observable event is emitted, the component sends a “communicate” message to all the components, and does not wait for an answer before continuing its execution.
3. When a loop is detected, the component sends a “loop” message to all the components.

**Matrix to record trajectories.** A component records its execution trajectories and their matched observed traces from the other components in a *matrix*. A cell contains an event and *component.id*. For convenience, the rows are aligned by the correspondent communication events (ref. Table 1 and Table 2).

**Distributed Simulation.** In Algorithm 1, a local model  $G_i = (Q_i, \Sigma_i, \delta_i, q_i^0)$  is simulated and the execution paths are recorded. The main While loop alternatively operates two methods *simulate()* and *processMessages()*, until all the paths are explored or some time-out conditions satisfied. The method *simulate()* simulates the local model by exploring all the possible execution paths. At each invocation, *simulate()* stops only when all the execution paths pause at the communication transitions waiting for confirmations, or all the execution paths go into stable loops. *processMessages()* is to process received messages.

As the local model has finite states, the whole simulation process runs into a loop eventually. We can detect a loop by comparing the states on the execution path. If a loop has no communication events, we just need to inform all the components that this loop is detected. There is no need to repeat the execution of the loop. If a loop contains communication events, we need to execute the loop again until all the components are in a stable and synchronized loop. An execution path is removed if its communication event is not confirmed after the “time-out” threshold. Therefore, Algorithm 1 terminates. We neglect some details in Algorithm 1 for simplicity.

**Algorithm 1** Distributed Simulation

$MUP, MP = \emptyset$  - the set of unprocessed/processed received messages.

$NUP, NP = \emptyset$  - the set of unprocessed/processed sent messages.

```

1:  $A = \{q_i^0\}$ ,  $\delta' = \emptyset$ 
2: while  $A \neq \emptyset$  or !(time-out) do
3:   simulate()
4:   processMessages()
5:    $A = \{\delta_i(q, e) | \delta_i \in \delta', \delta_i.pause = false\}$ 
6:    $\delta' = \delta' \setminus \{\delta_i | \delta_i \in \delta', \delta_i.pause = false\}$ 
7: end while
1: simulate()
2: while  $A \neq \emptyset$  do
3:    $\delta = \bigcup_{q \in A, e \in \Sigma} \delta(q, e, q')$ ,  $A' = \bigcup_{q \in A, e \in \Sigma} \delta(q, e)$ 
4:   for each transition  $\delta_i(q, e, q') \in \delta$  do
5:      $id = T.grows(\delta_i)$ ,  $Matrix \leftarrow (e, G_i.id)$ 
6:     if  $e \in \Sigma_c$  then
7:        $id.pause = true$ ,  $A' = A' \setminus q'$ ,  $\delta' \leftarrow \delta_i$ 
8:        $m = sends("communicate", \langle e, G_i.id, G_j.id \rangle)$  to  $\forall G_j \neq G_i, e \in G_j$ ,
9:        $NUP \leftarrow m$ 
10:    else
11:      if  $e \in \Sigma_o$  then
12:         $\forall G_j \neq G_i, m = sends("communicate", \langle e, G_i.id, G_j.id \rangle)$ 
13:      end if
14:    end if
15:    {Detect loops}
16:    if  $\exists \delta_j(q', e, q'') \in T.getPath(id) \wedge \delta_j.loop = false$  then
17:       $id' = \delta_j(q', e, q'').id$ 
18:       $\forall G_j \neq G_i, m = sends("loop", \langle G_i.id', G_i.id, G_j.id \rangle)$ 
19:       $E = \{e | e \in \delta_k, \delta_k \in (T.getPath(G_i.id) - T.getPath(G_i.id'))\}$ 
20:      mark the loop from  $id'$  to  $id$  in  $Matrix$ 
21:      if  $E \cap \Sigma_c = \emptyset \vee$  all aligned trajectories in loop then
22:         $A' = A' \setminus q'$ 
23:      end if
24:    end if
25:  end for
26: end while
1: processMessages()
2: for  $\forall m \in MUP$  do

```

```

3:   if  $m.type == "communication" \wedge m.e \in \Sigma_o$  then
4:     record  $(m.e, m.G_i.id)$  at the end of a line for  $G_j$  in  $Matrix$  where  $G_j == G_i \wedge G_i.id \leq G_j.id$ 
5:      $MUP = MUP \setminus m$ ,  $MP \leftarrow m^1$ 
6:   end if
7:   if  $m.type == "confirm" \wedge \exists m' \in NUP$  that  $m.match(m'.e, m'.G_i.id) == true$  then
8:     align  $(m.e, m.G_i.id)$  with  $(m'.e, m'.G_i.id)$  in  $Matrix$ ,  $MUP = MUP \setminus m$ ,  $MP \leftarrow m$ 
9:     if all correspondents aligned to  $m'$  then
10:        $m'.id.pause = false$ 
11:        $NUP = MUP \setminus m'$ ,  $NP \leftarrow m'$ 
12:     end if
13:   end if
14:   if  $m.type == "communication" \wedge m.e \in \Sigma_c$  then
15:     for  $\forall m' \in NUP$  that  $m.match(m'.e, m'.G_i.id) == true^2$  do
16:       align  $(m.e, m.G_i.id)$  with  $(m'.e, m'.G_i.id)$  in  $Matrix$ 
17:        $m'' = send("confirm", \langle m.e, m'.G_i.id, m.G_i.id \rangle)$ 
18:        $MUP = MUP \setminus m$ ,  $MP \leftarrow m$ 
19:       if all correspondents aligned to  $m'$  then
20:          $m'.id.pause = false$ 
21:          $NUP = MUP \setminus m'$ ,  $NP \leftarrow m'$ 
22:       end if
23:     end for
24:   end if
25: end for

```

**Example 2** Given two local models  $G_1$  and  $G_2$  as in Figure 1, the execution matrix are those displayed in Table 1 and Table 2. The execution trees are shown in Figure 4 (a) and (b). Please notice that the trajectory  $G_2.3$  has no matching trajectories in  $G_1$ . Thus the last row in Table 1 contains only the observable events of  $G_2.3$ . So does the last row in Table 2.  $\square$

**Simulation Complexity.** A local model  $G_i$  has maximally  $|Q_i|^2 * |\Sigma_i|$  transitions. From the code, the first for loop on line 4 in *simulate()* is executed maximally  $|Q_i|^2 * |\Sigma_i|$  times. We assume that the system is well synchronized and that, after  $n$  limited execution steps, the system is in a stable synchronized loop. This means the while loop on line 2 in *simulate()* is executed  $n$  times. Therefore, the time complexity of Algorithm 1 is  $O(|Q_i|^2 * |\Sigma_i| * n)$ . We have bounds for  $n$  in some simple cases. For example, when the model has only one loop at the “end” state (like in our example),  $n$  is the number of the transitions in the longest path between the initial state and the “end” state. Assume all the components are executed  $n$  times. The time complexity to simulate the whole system is  $O(\sum_i |Q_i|^2 * |\Sigma_i| * n)$ .

### 4.3 Checking Diagnosability

A component can use the following reasoning process to decide the diagnosability of the system. The conclusion by any component should be the same.

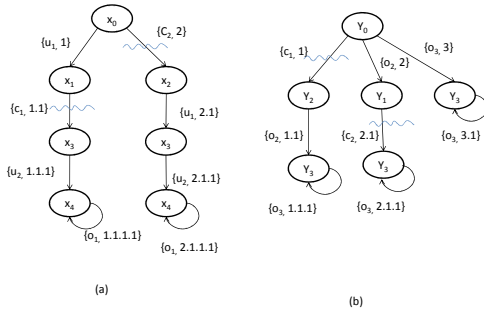
<sup>1</sup>reuse of  $MP$  and  $NP$  is eliminated

<sup>2</sup>if no match, a new path from the initial state may start.

Path $G_{1.1}$	$\{u_1, G_{1.1}\}$	$\{c_1, G_{1.1.1}\}, \otimes$	$\{u_2, G_{1.1.1.1}\}$	$\{o_1, G_{1.1.1.1.1}\}^*$	-	-
Path $G_{2.1}$	-	$\{c_1, G_{2.1}\}, \otimes$	$\{o_2, G_{2.1.1}\},$	$\{o_3, G_{2.1.1.1}\}^*$	-	✓ by $G_1, G_2$
Path $G_{1.2}$	-	$\{c_2, G_{1.2}\}, \otimes$	$\{u_1, G_{1.2.1}\}$	$\{u_2, G_{1.2.1.1}\}$	$\{o_1, G_{1.2.1.1.1}\}^*$	× by $G_1$
Path $G_{2.2}$	$\{o_2, G_{2.2}\}$	$\{c_2, G_{2.2.1}\}, \otimes$	$\{o_3, G_{2.2.1.1}\}^*$	-	-	✓ by $G_2$
Path $G_{2.3}$	$\{o_3, G_{2.3}\}$	$\{o_3, G_{2.3.1}\}^*$	-	-	-	✓ by $G_1, G_2$

Table 1: Matrix of execution trajectories of  $G_1$ .  $\otimes$  marks communication events, ✓ marks recognized trajectories.

Path $G_{2.1}$	$\{c_1, G_{2.1}\}, \otimes$	$\{o_2, G_{2.1.1}\}$	$\{o_3, G_{2.1.1.1}\}^*$	✓ by $G_1, G_2$
Path $G_{1.1}$	$\{c_1, G_{1.1.1}\}, \otimes$	$\{o_1, G_{1.1.1.1.1}\}^*$	-	-
Path $G_{2.2}$	$\{o_2, G_{2.2}\}$	$\{c_2, G_{2.2.1}\}, \otimes$	$\{o_3, G_{2.2.1.1}\}^*$	× by $G_1$
Path $G_{1.2}$	-	$\{c_2, G_{1.2}\}, \otimes$	$\{o_1, G_{1.2.1.1.1}\}^*$	✓ by $G_2$
Path $G_{2.3}$	$\{o_3, G_{2.3}\}$	$\{o_3, G_{2.3.1}\}^*$	-	✓ by $G_1, G_2$

Table 2: Matrix of execution trajectories of  $G_2$ .  $\otimes$  marks communication events, ✓ marks recognized trajectories.Figure 4: (a) The execution tree for component  $G_1$ ; (b) The execution tree for component  $G_2$ .

**1. The aligned local trajectories are globally consistent.** This is ensured by the design of the simulation process which follows the only constraint of the distributed system - synchronization by communication events. The aligned local trajectories can have sequential parts and concurrent parts. For example, Path  $G_{1.1}$  and Path  $G_{2.1}$  in Table 1 are paired and aligned by  $c_1$ . The pair can be represented by  $u_1c_1((u_2o_1^*)||o_2o_3^*)$ . The concurrent part joined by  $||$  indeed means that two FSMs synchronize without common events. The second trajectory in  $G_1$  formed by Path  $G_{1.2}$  and Path  $G_{2.2}$  can be represented by  $o_2c_2((u_1u_2o_1^*)||o_3^*)$ . And third trajectory in  $G_1$  is formed by Path  $G_{2.3}$  can be represented by  $o_3o_3^*$ . For  $G_2$ , the trajectory made by  $G_{2.1}$  and  $G_{1.1}$  is  $c_1((o_2o_3^*)||o_1^*)$ ; the trajectory made by  $G_{2.2}$  and  $G_{1.2}$  is  $o_2c_2(o_3^*||o_1^*)$ ; and the trajectory made by  $G_{2.3}$  is  $o_3.o_3^*$ .

**2. Each component can compute the global traces.** By projecting the aligned trajectories on observable events, a component can obtain global traces. For example, the group of traces for Path  $G_{1.1}$  and Path  $G_{2.1}$  is  $o_1^*||o_2o_3^*$ ; for Path  $G_{1.2}$  and Path  $G_{2.2}$  is  $o_2(o_1^*||o_3^*)$ , and for Path  $G_{2.3}$  is  $o_3o_3^*$ .  $G_2$  gets the same results.

**3. Compute the recognizability of the patterns.** First step: each component decides the recognizability of the local observed trajectories against its local pattern. We project each pair of trajectories on critical events to simplify the computation, because only the sequences of critical events decide recognizabil-

ity. For  $G_1$ , the three projected sequences for the three pairs of aligned trajectories are:  $u_1o_2o_3^*$ ,  $o_2(u_1||o_3^*)$ , and  $o_3o_3^*$ . It is easy to see that the first and the third are recognized against local pattern  $\Omega_1$ , and the second is rejected. For  $G_2$ , we have three sequences:  $o_2o_3^*$ ,  $o_2o_3^*$  and  $o_3o_3^*$ . All are recognized against local pattern  $\Omega_2$ . Please notice that, if a trajectory has concurrent terms, if and only if all the possible paths are recognized by the pattern, can we consider the trajectory is recognized. For example, assume the pattern is  $o_1o_2$  and the trajectory is  $o_1||o_2$ . This trajectory is considered non-recognizable, because one possible path  $o_2o_1$  is not recognizable by the pattern. The complexity to compute local recognition is discussed at the end of this section.

Second step: if and only if all the components unanimously vote for recognizing a pair of aligned trajectories using their local patterns, this pair of trajectories is recognized by the global pattern. That local recognition implies global recognition is ensured by Theorem 2. That a local rejection can cause a global rejection is easy to understand. If a global trajectory is recognized by the global pattern, its projection on one component's events can also satisfy this component's local pattern. Therefore, if the projection of a global trajectory on one component's events does not satisfy the component's local pattern, this global trajectory does not satisfy the global pattern either.

For our example, the pair of Path  $G_{1.1}$  and Path  $G_{2.1}$  is recognized globally; the pair of Path  $G_{1.2}$  and Path  $G_{2.2}$  is not recognized globally; and Path  $G_{2.3}$  is recognized globally.

**4. Decide diagnosability.** From Theorem 1, if there exist both recognized and unrecognized trajectories of the same trace, the system is not diagnosable, otherwise it is diagnosable. For our example, we find that  $o_2$  is a possible trace for the pair composed by Path  $G_{1.1}$  and Path  $G_{2.1}$  and the pair composed by Path  $G_{1.2}$  and Path  $G_{2.2}$ . As the pair of Path  $G_{1.1}$  and Path  $G_{2.1}$  are recognized by the global pattern and the pair Path  $G_{1.2}$  and Path  $G_{2.2}$  are not, the system is not diagnosable. As we check all the possible trajectories, our decision is sound. The complexity to check the existence of a common trace for trajectories is discussed at the end of this section.

Our reasoning process relies on a proposition that the local recognition of the local patterns implies

global recognition of the global pattern. It is easy to tell the inverse implication stands. As a local pattern is less constrained than the global pattern, if a local pattern is not recognized, the global pattern cannot be recognized. But there are cases that local recognition does not imply global recognition. Theorem 2 gives a sufficient condition that except the case that two consecutive critical transitions whose events are locally unobservable events and belong to two different components, local recognition implies global recognition.

**Theorem 3** *If there exist no two consecutive critical transitions in the pattern whose events are locally unobservable events and belong to different components, the locally recognized trajectories against the local patterns are globally recognized against the global pattern.*

**Proof:** Without losing generality, assume  $t_1$  and  $t_2$  are two consecutive critical transitions in the pattern and whose events are  $\alpha$  and  $\beta$  respectively.  $t_1 \prec t_2$  denotes that  $t_1$  precedes  $t_2$ .

If  $\alpha$  and  $\beta$  belong to the same component, it is a trivial case, because one component can fully determine their sequence from the simulation algorithm.

If  $\alpha \in G_1, \beta \in G_2$  belong to components  $G_1$  and  $G_2$  respectively. By the condition of the theorem,  $\alpha$  and  $\beta$  cannot be both locally unobservable events. Therefore,  $\alpha$  and  $\beta$  can be both locally observable, or one is locally unobservable and the other is locally observable. Thus, at least one of  $G_1$  and  $G_2$  has both  $\alpha$  and  $\beta$  in its local pattern. Therefore, if this component can determine their sequences locally through the simulation algorithm, this means globally their sequence is determined. Thus, the local recognition leads to the global recognition.  $\square$

Theorem 2 tells an exceptional case that if there are two consecutive critical transitions whose events are locally non-observable and belong to different components in the pattern, local recognition does not always imply global recognition. In this case, we cannot apply the methods developed in this paper. Figure 5 illustrates this exceptional case. In Figure 5(a), the pattern has two consecutive critical transitions whose events are  $u_1$  and  $u_2$  which belong to  $G_1$  and  $G_2$  respectively. Since one component can only simulate either  $u_1$  or  $u_2$ , there is no way to determine their sequence by either of the components. For example, Figure 5(c) and (d) illustrate two components  $G_1$  and  $G_2$ . Their local patterns are  $(p_0, u_1, p_1) \rightarrow (p_1, o_1, p_3)$  and  $(p_0, u_2, p_2) \rightarrow (p_2, o_1, p_3)$  respectively.

By distributed simulation,  $G_1$  gets one trajectory  $u_1 c_1 (o_1 || o_2)$  and  $G_2$  gets one trajectory  $u_2 c_1 (o_2 || o_1)$ . It is easy to see that both  $G_1$  and  $G_2$  recognize their local patterns. However, globally, we can only deduce  $u_1$  and  $u_2$  are parallel, i.e.  $u_1 || u_2$ , because they both precede the communication point  $c_1$ . There is no way to distinguish their sequential order. This can also be proved by synchronizing the local models of  $G_1$  and  $G_2$  to get the global system model. In this case, the local recognition does not lead to the global recognition.

If we change the pattern to Figure 5(b), where  $u_1$  and  $u_2$  are separated by an observable event  $o_1$ , the local patterns to  $G_1$  and  $G_2$  are  $(p_0, u_1, p_1) \rightarrow (p_1, o_1, p_3)$  and  $(p_0, o_1, p_2) \rightarrow (p_2, u_2, p_3)$  respectively.  $G_1$  can get  $u_1 c_1 (o_1 || o_2)$  by simulation, and it

recognizes its local pattern.  $G_2$  can get  $u_2 c_1 (o_1 || o_2)$  by simulation, and it does not recognize its local pattern. Therefore, globally, the pattern is not recognized. This is the case that satisfies the condition of Theorem 2. We can see that the sequence orders of  $u_1 \prec o_1$  and  $o_1 \prec u_2$  in the pattern can be determined by one component. Thus local recognition implies global recognition. However, our example is a case that local non-recognition implies global non-recognition.

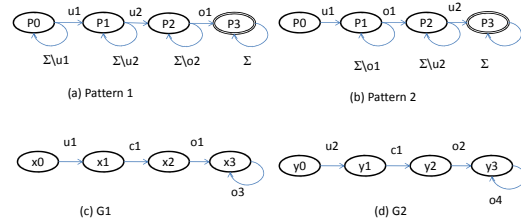


Figure 5: Two consecutive local events in a pattern.

Briefly, we discuss when a local component can determine the occurrence sequence of two critical events  $\alpha$  and  $\beta$  emitted by two consecutive critical transitions in a pattern. Here we do not constrain the observability of  $\alpha$  and  $\beta$ . If  $\alpha$  and  $\beta$  belong to the same component, then this component can determine their occurrence sequence by simulation. If  $\alpha$  and  $\beta$  belong to two different components  $G_1$  and  $G_2$  respectively, their occurrence sequence cannot be determined by their observed sequence. However, if there exists at least a communication event  $c$  shared by  $G_1$  and  $G_2$ , and  $\alpha \prec c, c \prec \beta$ , we can deduce that  $\alpha \prec \beta$ . This is illustrated as in Table 3. This helps us to check local recognition easily.

Path $G_1.1$	$\alpha$	$\rightsquigarrow$	$c \otimes$		
Path $G_2.1$			$c \otimes$	$\rightsquigarrow$	$\beta$

Table 3: Matrix of execution trajectories of  $G_1$  and  $G_2$ .

**Complexity of Checking Diagnosability.** The steps one and two in the diagnosability check reasoning process are trivial, which involve trajectories alignment and projection.

The third step needs to compute whether a pair of trajectories can be recognized by the local pattern. We first project the trajectories on critical events. In a general case, the projected trajectories have concurrent segments. Each projected trajectory can be represented as a regular expression. (Thompson, 1968) converts a regular expression  $E$  with  $n$  letters in length into a Nondeterministic Finite Automaton (NFA) with  $n$  states in linear time  $l(n)$ . The synchronized NFA  $\Pi$  has  $m = n_1 \cdot n_2$  states. Therefore, we need to consider whether  $L(\Pi) \subseteq L(\Omega_i)$ , where  $\Omega_i$  is a local pattern which is a Deterministic Finite Automaton (DFA). It is known that it takes  $O(n_1 \cdot n_2)$  time to check  $L_1 \subseteq L_2$ , for  $L_1$  and  $L_2$  are DFA with  $n_1$  and  $n_2$  states (Gelade and Neven, 2008). However, to convert a NFA  $\Pi$  to a DFA takes an exponential time  $2^m$ . Therefore, the total complexity is  $O(2^m \cdot p)$ , where  $p$  is the number of states in the pattern. Normally a pattern  $\Omega$  is relatively

simple and critical events are not many. We can think that  $m$  is much smaller than the length of trajectories.

The fourth step needs to compute the existence of common traces for two pairs of trajectories when one pair is recognized globally and the other is not. We first project the trajectories on observable events. Then we can get two synchronized NFAs  $\Pi_1$  and  $\Pi_2$  as above. We need to tell whether  $L(\Pi_1) \cap L(\Pi_2) = \emptyset$ . This needs  $O(m_1 \cdot m_2)$  time complexity (Gelade and Neven, 2008).

## 5 RELATED WORK

(Sampath *et al.*, 1995) introduced the diagnosability problem for discrete event systems and proposed to solve it by detecting some transition cycles of ambiguous states in a global diagnoser. The main disadvantage of their approach is its exponential space complexity in the number of system states. (Jiang *et al.*, 2001) proposed a classical twin plant approach to improve the algorithm complexity, which is only polynomial in the number of system states. (Pencolé, 2004) studied the diagnosability problem in distributed systems and provided a non scalable method of synchronizing local non reduced twin plants until a global critical path is detected. Then (Schumann and Pencolé, 2007) proposed a scalable approach for diagnosability verification in a distributed way through checking the existence of a set of local reduced twin plants, where at least one of them contains an observable possibly non-diagnosable cycle. On the other hand, (Jéron *et al.*, 2006) extended the diagnosability problem for fault events to that for supervision patterns, which can be used to describe more general objectives. They verified pattern diagnosability by employing a global twin plant method. (Ye *et al.*, 2009) is the first paper to discuss pattern diagnosability in distributed systems based on the model in (Jéron *et al.*, 2006). Their study limited to simple patterns which contain only one linear branch leading to the final state. Compared to (Ye *et al.*, 2009), our paper starts with the same model as in (Jéron *et al.*, 2006), but without any constraints on patterns. Thus, the result is more general than (Ye *et al.*, 2009). (Guillou *et al.*, 2008) studies diagnosis of discrete event system with a formalism called chronicle (Dousson *et al.*, 1993). A chronicle is a set of events and temporary constraints between those events. It is more abstract than pattern represented by FSM. (Guillou *et al.*, 2008) extends the classic chronicle to include variables for modeling data flow and to include synchronization points for using in distributed systems. In (Guillou *et al.*, 2008), faulty scenarios are modelled in the variables, while in our paper, the faults are presented by the patterns (or the FSM). In contrast to our paper, (Guillou *et al.*, 2008) studies the problem of diagnosis, instead of diagnosability. In addition, (Guillou *et al.*, 2008) uses local diagnosers and global diagnoser, while in our system we do not use any central node to synthesize local results.

## 6 CONCLUSIONS

We study pattern diagnosability in distributed discrete event systems. We present the principle of pattern diagnosability (Theorem 1). For a distributed system, we

use distributed simulation to identify globally consistent trajectories, and then determine if the trajectories with the same trace are uniformly recognized by the pattern or not. We present the correctness proof (Theorem 2) and complexity analysis.

## REFERENCES

- (Dousson *et al.*, 1993) Christophe Dousson, Paul Gaborit, and Malik Ghallab. Situation recognition: Representation and algorithms. In *IJCAI*, pages 166–174, 1993.
- (Gelade and Neven, 2008) Wouter Gelade and Frank Neven. Succinctness of the complement and intersection of regular expressions. In Susanne Albers and Pascal Weil, editors, *25th International Symposium on Theoretical Aspects of Computer Science (STACS 2008)*, volume 1, pages 325–336, Dagstuhl, Germany, 2008. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.
- (Guillou *et al.*, 2008) Xavier Le Guillou, Marie-Odile Cordier, Sophie Robin, and Laurence Rozé. Chronicles for on-line diagnosis of distributed systems. In *ECAI*, pages 194–198, 2008.
- (Jéron *et al.*, 2006) T. Jéron, H. Marchand, S. Pinchinat, and M.O. Cordier. Supervision patterns in discrete event systems diagnosis. *Proceedings of the 8th International Workshop on Discrete Event Systems*, July 2006.
- (Jiang *et al.*, 2001) S. Jiang, Z. Huang, V. Chandra, and R. Kumar. A polynomial time algorithm for diagnosability of discrete event systems. *IEEE Transactions on Automatic Control*, pages 46(8):1318–1321, 2001.
- (Pencolé, 2004) Y. Pencolé. Diagnosability analysis of distributed discrete event systems. *ECAI'04*, pages 43–47, 2004.
- (Sampath *et al.*, 1995) M. Sampath, R. Sengupta, S. Lafortune, K. Sinnamohideen, and D. Teneketzis. Diagnosability of discrete event system. *IEEE Transactions on Automatic Control*, pages 40(9):1555–1575, 1995.
- (Schumann and Pencolé, 2007) A. Schumann and Y. Pencolé. Scalable diagnosability checking of event-driven systems. *IJCAI-07*, pages 575–580, 2007.
- (Thompson, 1968) Ken Thompson. Regular expression search algorithm. *Communications of the ACM*, 6(11), June 1968.
- (Ye *et al.*, 2009) Lina Ye, Philippe Dague, and Yuhong Yan. Pattern diagnosability in distributed discrete event systems. In *Proceedings of the 20th International Workshop on Principles of Diagnosis*, pages 179–186, Stockholm, Sweden, June 2009.