Preprints of the 21st IFAC World Congress (Virtual)
Berlin, Germany, July 12-17, 2020

Synchronism Recovery of Discrete Event Systems ⋆

Lucas V. R. Alves * **Patrícia N. Pena** **

* *Technical College and the Graduate Program in Electrical Engineering Universidade Federal de Minas Gerais Belo Horizonte, Brazil lucasvra@ufmg.br*
** *Electronics Engineering Department Universidade Federal de Minas Gerais Belo Horizonte, Brazil ppena@ufmg.br*

Abstract: Most of the systems are build of components that should stay synchronized for the system to work properly. Usually, the synchronism of these subsystems is maintained through communication and this communication is subject to failures, leading the system to a undesirable state, where the states of the components do not match. In this sense, this paper deals with the problem of resynchronizing components of a system, leading to a global state where the individual states of the subsystems match with each other. In order to do so, an algorithm, using ideas of synchronizing automata, automata that reach a specific state when a synchronizing word is executed, regardless the origin state, is presented.

Keywords: Discrete Event Systems, Recovery, Synchronizing Automata

1. INTRODUCTION

Fault recovery is an important issue in modern industries. Once a fault has happened, the desynchronization of the components may follow. This problem can happen in industrial systems, for example, when the controller loses observation on the plant, in aerospace systems, when the mission control loses, for some time, communication with a satellite, in communication systems, when two pieces of software, interacting through the network, temporarily lose communication and many other situations.

Usually these problems, in discrete event systems modelled as automata, lead to a situation where the state of the components of the system do not match, leading to an incorrect behavior. The solution to theses problems are not always straightforward and, in many situations it is not possible to just restart the system (Abad et al., 2016).

In this work we deal with the problem of recovering desynchonized Discrete Events Systems (DES), leading the system to a safe operation, where the states of all the components match with each other. In order to do so, some assumptions must be obeyed regarding the operation of the system. Such assumptions are introduced later.

The error recovery problem in Discrete Event Systems can be divided into three sub-problems (Loborg, 1994):

(1) Detection: Consists in detecting discrepancies between the state of the system components (Carvalho et al., 2018).
(2) Diagnostic: Consists in detecting the fault that generated the discrepancy. In DES, this problem may be handled using techniques of diagnosability of Discrete Event Systems (Lafortune et al., 2018).
(3) Recovery: After eliminating the cause of the fault, the recovery may be about changing the state of the system and supervisor to be consistent.

There are several approaches to deal with recovery in Discrete Event Systems, most of then in the industrial field. Shu (Shu, 2014) deals with the problem of recovery in systems using *recovery events*, fired when the system reaches a failure state. On the other hand, Andersson and coauthors (Andersson et al., 2009, 2010, 2011), Bergagard and coauthors (Bergagrd and Fabian, 2013; Bergagrd et al., 2015) present a method to recover manufacturing systems, modelled by operations and coordination of operations (COP), using the concept of *restart states*, allowing the recovery process to lead the physical plant and the COP to matching states.

This paper proposes a general method for the recovery of Discrete Event Systems based on the search for a single sequence that, regardless the faulty state the system is, will always lead the system to a safe state.

If the system components are modeled by synchronizing automata (Volkov, 2008) then it is always possible to lead the system to a known state. An automaton is said to be *synchronizing* when there is a word, called *synchronizing word*, such that, when executed by the automaton, regardless of the state of origin, leads the automaton to the same destination state. So, two identical automata, in different states (desynchonized), will always evolve to the same state when a *synchronizing word* is executed (Volkov, 2008), becoming synchronized again.

⋆ This work has been supported by the Brazilian agency CAPES, the National Council for Scientific and Technological Development CNPq grant 443656/2018 − 5 and Fapemig.

The existence of a synchronizing word has applications in many fields, such as robotics, assembling, loading and packing of products (Natarajan, 1986, 1989). More theoretical development was presented in the context of industrial automation (Eppstein, 1988; Goldberg, 1993; Chen and Ierardi, 1995). Synchronizing automata were also applied to problems with partial observability (Larsen et al., 2014) and problems modeled with Petri Nets (Pocci et al., 2013, 2014a,b, 2016).

In fact, synchronizing automata may solve desynchronization problems not only when the components of the system are identical, but also when they share at least a synchronizing word (Benenson et al., 2003). Typically, each component of the system have a distinct automaton model such that, even if all the components are synchronizing automata they may not share a synchronizing word.

We propose a two-step method, where first we turn each component of the system into a complete automaton, with complete transition function, by adding self-loops, and then we use an algorithm to search for a *resynchronizing word* which leads the system to a synchronizing state.

The *resynchronizing word* differs from the original synchronization concept, because, instead of leading the system to a single state, it leads the system to tuples of states composed by *safe states*, where all system components are synchronized.

This paper is organized such that Section 2 has the preliminaries, where we show the main concepts needed to understand the results. Section 3 presents the main results. In Section 4, an example is presented showing the methodology being applied to an industrial system. The conclusions are in Section 5.

2. PRELIMINARIES

In this section, we summarize some fundamental concepts of Discrete Event Systems modelled as automata, that are needed for the theoretical development of the paper. We, also, define some concepts and notation on *synchronizing automata*.

2.1 Languages and Automata

Let Σ be a finite nonempty set of *events*, referred to as an *event set*. Behaviors of DES are modeled by finite words over Σ. The *Kleene closure* Σ^* is the set of all words on Σ, including the empty word ε. A subset $L \subseteq \Sigma^*$ is called a *language*. The *concatenation* of words $s, u \in \Sigma^*$ is written as su. A word $s \in \Sigma^*$ is called a *prefix* of $t \in \Sigma^*$, written $s \leq t$, if there exists $u \in \Sigma^*$ such that $su = t$. The *prefix-closure* $\overline{L}$ of a language $L \subseteq \Sigma^*$ is the set of all prefixes of words in L, i.e., $\overline{L} = \{\, s \in \Sigma^* \mid s \leq t \text{ for some } t \in L \,\}$.

A common operation over words and languages is the natural projection. Given two event sets Σ and Σ_i, such that $\Sigma_i \subseteq \Sigma$, the natural projection $P_{\Sigma\to\Sigma_i} : \Sigma^* \to \Sigma_i^*$ is defined as:

$$P_{\Sigma\to\Sigma_i}(\epsilon) = \epsilon$$
$$P_{\Sigma\to\Sigma_i}(\sigma) = \begin{cases}\sigma & \text{if } \sigma \in \Sigma_i \\ \epsilon & \text{if } \sigma \in \Sigma \setminus \Sigma_i\end{cases}$$
$$P_{\Sigma\to\Sigma_i}(s\sigma) = P_{\Sigma\to\Sigma_i}(s)P_{\Sigma\to\Sigma_i}(\sigma) \text{to } s \in \Sigma^*, \sigma \in \Sigma.$$

The inverse projection maps a word built from an event set Σ_i to a language in the event set Σ as:

$$P^{-1}_{\Sigma\to\Sigma_i}(t) = \{s \in \Sigma^* \mid P_{\Sigma\to\Sigma_i}(s) = t\}.$$

Both operations can be extended to operate over languages. For $L \subseteq \Sigma^*$:

$$P_{\Sigma\to\Sigma_i}(L) = \{t \in \Sigma_i^* \mid (\exists s \in L)\,[P_{\Sigma\to\Sigma_i}(s) = t]\}.$$

For $L \subseteq \Sigma_i^*$:

$$P^{-1}_{\Sigma\to\Sigma_i}(L) = \{s \in \Sigma^* \mid (\exists t \in P_{\Sigma\to\Sigma_i}(L))\,[P_{\Sigma\to\Sigma_i}(s) = t]\}.$$

Definition 1. *A Deterministic Finite Automata (DFA) is a 5-tuple $G = (Q, \Sigma, \delta, q_0, Q_m)$, where Q is a finite set of states, Σ is an event set, $\delta : Q \times \Sigma \to Q$ is the transition function, $q_0 \in Q$ is the initial state and $Q_m \subseteq Q$ is the set of marked states.* □

The transition function can be extended to recognize words over Σ^* as $\delta(q, \sigma s) = q'$ with $\delta(q, \sigma) = x$ and $\delta(x, s) = q'$.

The execution of a word s in a state q, $\delta(q, s)$, is denoted by the concatenation $q.s$. The same notation is used to represent this operation over sets. The notation $A.s$ denotes the set of destination states when the word s is executed from the set of states $A \subseteq Q$.

The active event function, defined by $\Gamma : Q \to 2^\Sigma$, is, given a state q, the set of events $\sigma \in \Sigma$ for which $\delta(q, \sigma)$ is defined.

The generated and marked languages are, respectively, $\mathcal{L}(G) = \{s \in \Sigma^* | q_0.s = q' \wedge q' \in Q\}$ and $\mathcal{L}_m(G) = \{s \in \Sigma^* | q_0.s = q' \wedge q' \in Q_m\}$. Another language is defined to include words starting in any state q of G as $\mathcal{L}_G(q) = \{s \in \Sigma^* | q.s = q' \wedge q, q' \in Q\}$ such that $\mathcal{L}_G(q_0) = \mathcal{L}(G)$. An automaton is said to be nonblocking if $\overline{\mathcal{L}_m(G)} = \mathcal{L}(G)$.

Definition 2. *Let $G_1 = (Q_1, \Sigma_1, \delta_1, q_{01}, Q_{m1})$ and $G_2 = (Q_2, \Sigma_2, \delta_2, q_{02}, Q_{m2})$ be two automata. The parallel composition of G_1 and G_2, denoted by $G_{12} = G_1||G_2$ is:*

$$G_{12} = (Q_1 \times Q_2, \Sigma_1 \cup \Sigma_2, \delta_{12}, (q_{01}, q_{02}), Q_{m1} \times Q_{m2})$$

where

$$\delta((q_1, q_2), \sigma) = \begin{cases}(\delta_1(q_1, \sigma), \delta_2(q_2, \sigma)), & \text{if } \sigma \in \Gamma_1(q_1) \cap \Gamma_2(q_2) \\ (\delta_1(q_1, \sigma), q_2), & \text{if } \sigma \in \Gamma_1(q_1) \backslash \Sigma_2 \\ (q_1, \delta_2(q_2, \sigma)), & \text{if } \sigma \in \Gamma_2(q_2) \backslash \Sigma_1 \\ \text{undefined}, & \text{otherwise.}\end{cases}$$

Also, let $P_{\Sigma_1\cup\Sigma_2\to\Sigma_1} : (\Sigma_1 \cup \Sigma_2)^* \to \Sigma_1^*$ and $P_{\Sigma_1\cup\Sigma_2\to\Sigma_2} : (\Sigma_1 \cup \Sigma_2)^* \to \Sigma_2^*$ be natural projections:

$$\mathcal{L}(G_{12}) = P^{-1}_{\Sigma_1\cup\Sigma_2\to\Sigma_1}(\mathcal{L}(G_1)) \cap P^{-1}_{\Sigma_1\cup\Sigma_2\to\Sigma_2}(\mathcal{L}(G_2))$$
$$\mathcal{L}_m(G_{12}) = P^{-1}_{\Sigma_1\cup\Sigma_2\to\Sigma_1}(\mathcal{L}_m(G_1)) \cap P^{-1}_{\Sigma_1\cup\Sigma_2\to\Sigma_2}(\mathcal{L}_m(G_2)).$$

□

2.2 Synchronizing Automata

A synchronizing deterministic finite automaton is a DFA that has a word that, when executed from any state of the automaton, leads to a known state.

Definition 3. *(Volkov, 2008) A complete automaton $G = (Q, \Sigma, \delta, \_, \_)$ is synchronizing if and only if for any pair of states $q, q' \in Q$ there exists a word $w \in \Sigma^*$, called synchronizing word, such that $q.w = q'.w$, $\forall q, q' \in Q$.* □

A *complete automaton* in the definition refers to an automaton with a complete transition function, that is, transitions labeled with all the events in the event set are available in each state. Also, the initial state is irrelevant to the property, so it is intentionally omitted in the following example.

Example 1. *Consider the synchronizing automaton* $A = (Q, \Sigma, \_, \_, \_)$ *of Fig. 1. The word* $w = ab^3ab^3a$ *leads the automaton to state 1, regardless the origin state. Using the notation established before,* $Q.w = 1$, $Q = \{0, 1, 2, 3\}$. *It is straightforward that any word* sw, $s \in \Sigma^*$, *also leads the automaton to state 1.*

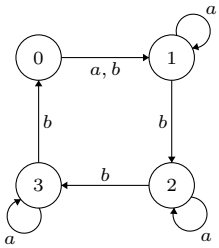


Fig. 1. Example 1- Conventional synchronizing automaton (Volkov, 2008).

□

If the word w is a synchronizing word, the operation $Q.w$ results in a singleton set. Also, the set of all synchronizing words of an automaton G is denoted by $Syn(G)$:

$$Syn(G) = \{w \in \Sigma^* | \ |Q.w| = 1\}.$$

3. DEVELOPMENT

The main idea of this work is to adapt the theory of synchronizing automata to the situation in which there are multiple distinct automata that need to be synchronized. To do so, instead of finding a synchronizing word which leads to a single destination state regardless the origin state, we want to find a *resynchronizing word* that leads to a state synchronizing tuple (one state for each component of the system) regardless of the original state tuple.

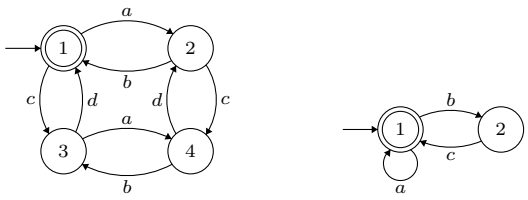
It is very usual that the automaton models of a system have partial transition function, so some events are forbidden to occur in some states. In order to apply the concepts already defined, it is necessary to complete the transition function of the automata. To do so, consider that, the forbidden events are self-loops in the automaton and, in practice are ignored.

Definition 4. *Let* $G = (Q, \Sigma, \delta, q_0, Q_m)$ *be the automaton model of a component of the system and* δ *be a partial transition function. We can define a new complete automaton* $G_c = (Q, \Sigma_{G_c}, \delta_c, q_0, Q_m)$, *with* $\Sigma \subseteq \Sigma_{G_c}$, *where:*

$$\delta_c(q, \sigma) = \begin{cases} q, & \delta(q, \sigma) \text{ is undefined} \\ \delta(q, \sigma), & \text{otherwise} \end{cases} \quad (1)$$

If follows from this operation that $\mathcal{L}(G_c) = \Sigma^*_{G_c}$. *Usually,* $\Sigma_{G_c} = \bigcup_{i=1}^{n} \Sigma_i$ *and* Σ_i *is the event set of the i-th system component.* □

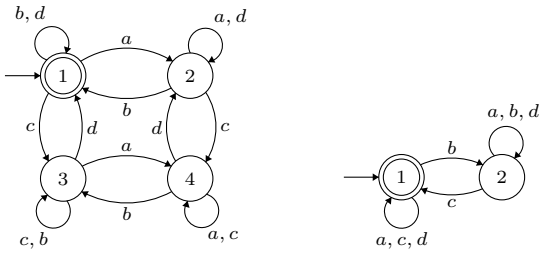
Example 2. *Consider a system composed by two automata,* G_1 *and* G_2 *presented in Fig. 2. Both automata are not complete and the event set is* $\Sigma = \{a, b, c, d\}$. *The automata shown in Fig.2 are neither complete nor have the same event set. The procedure used to turn the automaton*



(a) Original G_1 (b) Original G_2

Fig. 2. The components of the system

into a complete automaton (Fig. 3) consists on adding self-loops to each state with the missing events. These self-loops indicate that the system components ignore events that are not supposed to happen in the state, and their occurrence does not cause problems to the execution of the system.



(a) Automaton G_1 (b) Automaton G_2

Fig. 3. The components of the system with complete transition function

□

It is important to note that, to apply this transformation, the automata models of the system must be robust so they do maintain a correct behavior even when forbidden events occur and the occurrence of these events are ignored by the real system.

Definition 5. *Let* $G_i = (Q_i, \Sigma, \delta_i, \_, \_)$, $i \in \{1 \dots m\}$, *be the components of the system, with the same event set* (Σ) *and complete* ($\mathcal{L}(G_i) = \Sigma^*$). *The complete behavior of the system can be modelled by the automaton* $T = ||_{i=1}^{m} G_i = (Q_T, \Sigma, \delta_T, \_, \_)$. □

Each state of T is a tuple of states $(q_1, q_2, \dots, q_m)$, such that $q_i \in Q_i$. If T is a synchronizing automaton, it is possible to find a synchronizing word that leads the system to a state where all the states of the system are synchronized. In this work we do not care to which state of T the system goes, only if in the destination state is in Q_s, so instead of finding a sequence that leads the system to unique state, we expect to find a sequence that leads the system to any state in which the system components are synchronized.

Definition 6. *Let* $T = (Q_T, \Sigma, \delta_T, \_, \_)$ *be an automaton that models the complete behavior of the system, a state set* $Q_s \subset Q_T$ *is called a synchronizing set when for all states in* Q_s, *the correspondent states of the system components are synchronized. On the other hand, a state set* $Q_f = Q_T \setminus Q_s$ *is called failure set if the states of the system components are not synchronized.* □

Each state in Q_T is a tuple of states of the n components of the systems, namely, $q_T = \delta_T(q_{0T}, s) = (\delta_1(q_{01}, s), \delta_2(q_{02}, s), \dots, \delta_n(q_{0n}, s)) = (q_1, q_2, \dots, q_n)$.

Example 3. *The automaton* T *for the system presented in Example 2 is shown in Fig. 4. The states colored in gray are the states in the synchronizing set* (Q_s). *In a normal operation, the system will never reach a state outside* Q_s.

However, may a failure happen, the two automata will lose synchronization and a state in Q_f is going to be reached. In

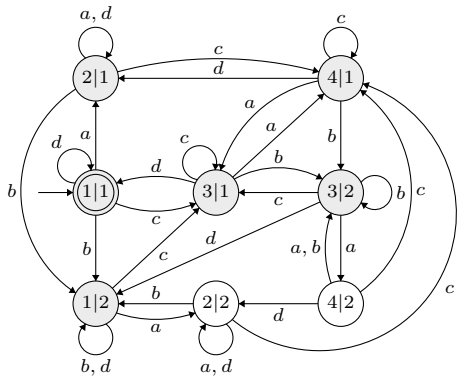


Fig. 4. Automaton T for the system presented in Example 2

this case, as the automata components are those in Fig. 2, we defined Q_s as the state set of the composition of the automata G_1 and G_2 of Fig. 2. The other states in Q_T should not be reached in normal conditions. □

Our objective is to find a *resynchronizing word* w_r in the automaton T, such that $Q_T.w_r \subseteq Q_s$. To do so, we present a heuristic algorithm that searches for at least one *resynchronizing word*. The algorithm is not guaranteed to succeed given that the existence of such word depends on the topology of the system.

3.1 Algorithm

Given that we have the automaton T which models the complete behavior of the system, it is necessary to find a sequence w_r that always leads to a state in Q_s regardless the state of the system. To do so, we represent the system as a *powerset automaton*:

Definition 7. *Let $T = (Q_T, \Sigma, \delta_T, q_0, Q_{mT})$ be a complete automaton which models the complete behavior of the system. We can define a new complete automaton $P = (Q_P, \Sigma, \delta_P, q_{0P}, Q_{mP})$ where:*

$$Q_P = 2^{Q_T} \setminus \emptyset,$$
$$Q_{mP} = 2^{Q_s} \setminus \emptyset,$$
$$\delta_P(q_P, \sigma) = \{\delta_T(q, \sigma) \mid q \in q_P \wedge q_P \in Q_P\}.$$

□

Example 4. *The powerset automaton P for the automaton T, presented in Example 3, is and automaton with 18 states, and 9 of then are marked. Reaching one of these states leads the system to resynchronization. For space limitation, only part of automaton P is shown in Fig. 5, the marked states are denoted in gray.*

It is important to note that, as the automaton T has 12 states, the automaton P has, in the worse scenario, $2^{12} - 1 = 2047$ states, but the algorithm will only evaluate 42 of then, given that it is only necessary to use the accessible part of P. □

In this sense, any $w_r \in \mathcal{L}_m(P)$ leads the system to a safe state because it leads to a marked state of P, so our algorithm only has to find a word that leads the system to a marked state.

Algorithm 1: Resynchonization Word Search

Data: $P = (Q_P, \Sigma, \delta_P, q_{0P}, Q_{mP})$
Result: w_r
frontier $\leftarrow \{q_{0P}\}$
path$[q_{0P}] \leftarrow \epsilon$
visited $\leftarrow \emptyset$

while $|frontier| > 0$ **do**
  visited $\leftarrow$ visited $\cup$ frontier
  new_frontier $\leftarrow \emptyset$
  foreach $q_o \in$ *frontier* **do**
    $match_o \leftarrow |q_o \cap Q_s|/|q_o|$
    foreach $\sigma \in \Sigma$ **do**
      $q_d \leftarrow \delta_P(q_o, \sigma)$
      $match_d \leftarrow |q_d \cap Q_s|/|q_d|$
      if $q_d \in$ *visited OR* $match_d > match_o$ **then**
        continue
      path$[q_d] \leftarrow$ path$[q_o]$ σ
      if $q_d \in Q_{mP}$ **then**
        $w_r \leftarrow$ path$[q_d]$
        return
      new_frontier $\leftarrow$ new_frontier $\cup\{q_d\}$
  frontier $\leftarrow$ new_frontier

The algorithm consists on an *Breadth First Search*, starting at the initial state of P and visiting each state of P until a marked state is reached. The set *visited* has the states already visited by the algorithm, to avoid that the same state is evaluated multiple times given that the automaton is, usually, cyclic. The structure *path* holds the sequence that leads to each visited state.

The first frontier is the initial state, composed of all states of T, and the *path* to the initial state is ϵ (the empty sequence). After the initialization, at each iteration the algorithm builds a new frontier composed by the adjacent states that were not visited yet.

For each state, the algorithm evaluates its *match* value, a percentage measure of how many of the states of T in the current state (in P) belong to Q_s. The *match* value varies from 0%, when none of the states belong to Q_s, to 100%, when all states belong to Q_s

The heuristic step of the algorithm consists on only adding a state to the frontier if it has the same or greater *match* value. This heuristic makes sense since it is the variable we want to maximize (leading to 100%), but there is no guarantee that a decrease in the match value cannot lead to a faster increase after. If the algorithm is applied without the heuristic it will find the shortest resynchonization word, when it exists, but when the heuristic is applied there is no guarantee that a sequence will be found or that it will be the shortest.

The algorithm stops when a state with 100% *match* is reached, returning the first sequence w_r found.

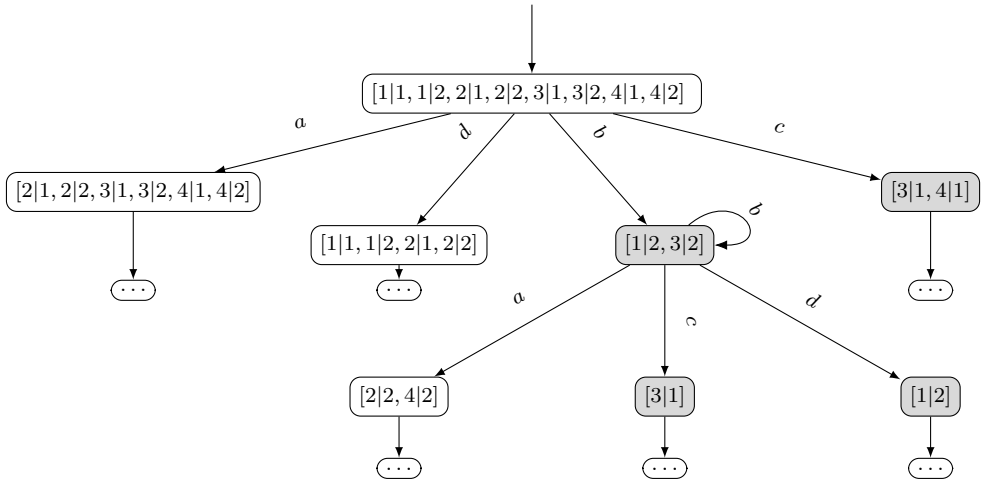


Fig. 5. First levels of the automaton P of the system presented in Example 2

Example 5. *Applying the algorithm to the automaton P presented in Fig. 5, the algorithm starts at state* $[1|1, 1|2, 2|1, 2|2, 3|1, 3|2, 4|1, 4|2]$ *and stops at the first iteration when it reaches one of the marked states* $[1|2, 3|2]$ *or* $[3|1, 4|1]$. *The sequence $w_r = b$ is one of the sequences that is obtained by the algorithm, and when executed in the automaton T (Fig. 4), regardless the starting state always leads to state $1|2$ or state $3|2$, both in Q_s so, meaning that in these states the components of the system are synchronized. The match value for each state presented in Fig. 5 is shown in Table 1.*

Table 1. Match value calculated for the states in Fig. 5

State	Match
$[1\|1, 1\|2, 2\|1, 2\|2, 3\|1, 3\|2, 4\|1, 4\|2]$	75%
$[2\|1, 2\|2, 3\|1, 3\|2, 4\|1, 4\|2]$	63%
$[1\|2, 3\|2]$	100%
$[3\|1, 4\|1]$	100%
$[1\|1, 1\|2, 2\|1, 2\|2]$	75%
$[2\|2, 4\|2]$	0%
$[3\|1]$	100%
$[1\|2]$	100%

3.2 Implementation Notes

In the worse case, when the heuristic has no effect on reducing the branch factor, the time complexity of the algorithm is $\mathcal{O}(|Q_P| + |\Sigma|)$, but it is important to note that Q_P is usually very large, given that it represents all the combinations of states of the components. in this sense, it is necessary to be careful on the implementation of the algorithm.

The automaton P should not be computed at once, but on-the-fly during the execution of the algorithm such that the large number of states in Q_P will not have to be stored in the memory. Other simplification is to limit the size of the frontier, reducing the branching factor of the algorithm, so after the next frontier if reaches the maximum allowed size, the states in the current frontier are not evaluated anymore.

4. CASE STUDY

As an example of application, we are going to use an industrial system called *Linear Cluster Tool* (Su et al., 2012), where the Supervisory Control Theory (Ramadge and Wonham, 1989) is applied in order to create a controller (supervisor) for the system. This system is extensible, by adding clusters to the system, allowing to test the method in small to medium size systems.

The Linear Cluster Tool , Fig. 6 consists on processing chambers $(C_1, C_2, \ldots, C_n)$, robots $(R_1, R_2, \ldots, R_n)$, unit buffers between the robot and the chamber $(B_1, B_2, \ldots, B_n)$, and between the robots $(B_{1|2}, B_{2|3}, \ldots, B_{n-1|n})$. Neither the automata of the plants nor the automaton of the supervisor are complete, so we apply the method presented in Definition 4.

Each state of the system consists on a tuple $q = (q_p, q_c)$, $q_p \in Q_p, q_c \in Q_c$ where Q_p is the state set of the plant and Q_c is the state set of the controller. Some of the pairs $Q_s \subseteq Q_p \times Q_c$ indicate the correct behavior of the system, but when $q \notin Q_s$ the system is in a failure state and, so, the correct control action is not being applied to the system, leading to a deadlock or allowing that incorrect actions are executed. In this situation, when possible the presented algorithm could be executed in order to find a sequence that, regardless the state of the system, always leads to a state in Q_s.

We applied the algorithm for the plant and controller (supervisor) of Cluster Tools with 2, 3, 4 and 5 robots and the results are shown in Table 2.

Table 2. Algorithm Execution for the Cluster Tool

Robots	States in the Plant	States in the Controller	Length of w_r	Algorithm Exec. Time
2	48	45	07 events	0.15 sec.
3	384	419	11 events	11.21 sec.
4	3,072	4,184	15 events	260.21 sec.
5	24,576	42,964	19 events	46,147.23 sec.

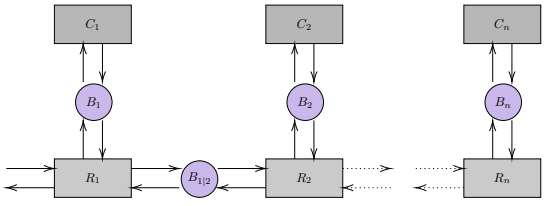


Fig. 6. Cluster Tool Diagram

As we can see, the algorithm is able to solve small to medium size problems, when given enough time. The size of automaton P is the main limitation of the algorithm, for the Cluster Tool with 5 robots, the upper bound of states in P is given as $2^{24,576 \times 42,964}$. The tests were made in a server with an Intel Xeon E5-2470 processor, at $2.4GHz$, and $96GB$ of RAM memory.

5. CONCLUSIONS

This paper presents an algorithm to resynchronize automata components of a system when a fault occurs leading the system to a state where the states of the components of the system do not match. In order to do so, the models of the components need to be complete automata or, at least, they need to be robust to the execution of events forbidden in their states, what allow us to model these automata as complete.

We presented a case study, showing each step of the method applied to a *Linear Cluster Tool*, a system in the industrial field. The same technique is applicable in any field if the assumptions made hold.

The complexity of the algorithm is linear, but we apply nonlinear transformations on the original automata an so, it is necessary to deal with state explosion on the automaton P. This is illustrated in the example, where a pair of automata with 4 and 6 states end up in an automaton with 113 states. The algorithm is applicable to real life problems, but, in some cases, some additional heuristics must be applied.

As a future research in this topic, we pretend to modularize this method, so, instead of working with one large automaton P, we could work with a set of smaller automata in parallel.

REFERENCES

Abad, F., Mancuso, R., Bak, S., Dantsker, O., and Caccamo, M. (2016). Reset-based recovery for real-time cyber-physical systems with temporal safety constraints. In *2016 IEEE 21st International Conference on Emerging Technologies and Factory Automation (ETFA)*, 1–8.

Andersson, K., Lennartson, B., and Fabian, M. (2010). Restarting manufacturing systems; restart states and restartability. *IEEE Transactions on Automation Science and Engineering*, 7(3), 486–499.

Andersson, K., Lennartson, B., Falkman, P., and Fabian, M. (2011). Generation of restart states for manufacturing cell controllers. *Control Engineering Practice*, 19(9), 1014 – 1022. Special Section: DCDS09 The 2nd IFAC Workshop on Dependable Control of Discrete Systems.

Andersson, K., Lennartson, B., and Fabian, M. (2009). Synthesis of restart states for manufacturing cell controllers. *IFAC Proceedings Volumes*, 42(5), 263–268.

Benenson, Y., Adar, R., Paz-Elizur, T., Livneh, Z., and Shapiro, E. (2003). Dna molecule provides a computing machine with both data and fuel. *Proceedings of the National Academy of Sciences*, 100(5), 2191–2196.

Bergagrd, P. and Fabian, M. (2013). Calculating restart states for systems modeled by operations using supervisory control theory. *Machines*, 1(3), 116–141.

Bergagrd, P., Falkman, P., and Fabian, M. (2015). Modeling and automatic calculation of restart states for an industrial windscreen mounting station. *IFAC-PapersOnLine*, 48(3), 1030 – 1036. 15th IFAC Symposium on Information Control Problems in Manufacturing.

Carvalho, L.K., Wu, Y.C., Kwong, R., and Lafortune, S. (2018). Detection and mitigation of classes of attacks in supervisory control systems. *Automatica*, 97, 121 – 133.

Chen, Y.B. and Ierardi, D. (1995). The complexity of oblivious plans for orienting and distinguishing polygonal parts. *Algorithmica*, 14(5), 367–397.

Eppstein, D. (1988). Reset sequences for finite automata with application to design of parts orienters. In *Proceedings of the 15th International Colloquium on Automata, Languages and Programming*, ICALP '88, 230–238. Springer-Verlag, London, UK.

Goldberg, K.Y. (1993). Orienting polygonal parts without sensors. *Algorithmica*, 10(2), 201–225.

Lafortune, S., Lin, F., and Hadjicostis, C.N. (2018). On the history of diagnosability and opacity in discrete event systems. *Annual Reviews in Control*, 45, 257–266. doi: 10.1016/j.arcontrol.2018.04.002.

Larsen, K.G., Laursen, S., and Srba, J. (2014). Synchronizing strategies under partial observability. In *International Conference on Concurrency Theory*, 188–202. Springer.

Loborg, P. (1994). Error recovery in automation an overview. *AAAI Spring Symposium on Detecting and Resolving Errors in Manufacturing Systems*, 94–100.

Natarajan, B.K. (1986). An algorithmic approach to the automated design of parts orienters. In *Proceedings of the 27th Annual Symposium on Foundations of Computer Science*, SFCS '86, 132–142. IEEE Computer Society, Washington, DC, USA.

Natarajan, B. (1989). Some paradigms for the automated design of parts feeders. *The International Journal of Robotics Research*, 8(6), 98–109.

Pocci, M., Demongodin, I., Giambiasi, N., and Giua, A. (2013). A new algorithm to compute synchronizing sequences for synchronized petri nets. In *TENCON 2013-2013 IEEE Region 10 Conference (31194)*, 1–6. IEEE.

Pocci, M., Demongodin, I., Giambiasi, N., and Giua, A. (2014a). Testing experiments on synchronized petri nets. *IEEE Transactions on Automation Science and Engineering*, 11(1), 125–138.

Pocci, M., Demongodin, I., Giambiasi, N., and Giua, A. (2014b). Testing experiments on unbounded systems: synchronizing sequences using petri nets. *IFAC Proceedings Volumes*, 47(2), 155–161.

Pocci, M., Demongodin, I., Giambiasi, N., and Giua, A. (2016). Synchronizing sequences on a class of unbounded systems using synchronized petri nets. *Discrete Event Dynamic Systems*, 26(1), 85–108.

Ramadge, P.J.G. and Wonham, W.M. (1989). The Control of Discrete Event Systems. *Proc. of the IEEE*, 77(1), 81–98.

Shu, S. (2014). Recoverability of discrete-event systems with faults. *IEEE Transactions on Automation Science and Engineering*, 11(3), 930–935.

Su, R., van Schuppen, J.H., and Rooda, J.E. (2012). The synthesis of time optimal supervisors by using heaps-of-pieces. *IEEE Transactions on Automatic Control*, 57(1), 105–118.

Volkov, M.V. (2008). *Synchronizing Automata and the Černý Conjecture*, 11–27. Springer Berlin Heidelberg, Berlin, Heidelberg.