DIAGNOSER REDUCTION OF DISCRETE-EVENT SYSTEMS

Augusto Pedro Mendonça de Vasconcellos

Dissertação de Mestrado apresentada ao Programa de Pós-graduação em Engenharia Elétrica, COPPE, da Universidade Federal do Rio de Janeiro, como parte dos requisitos necessários à obtenção do título de Mestre em Engenharia Elétrica.

Orientadores: Marcos Vicente de Brito Moreira
Gustavo da Silva Viana

Rio de Janeiro
Março de 2020

DIAGNOSER REDUCTION OF DISCRETE-EVENT SYSTEMS

Augusto Pedro Mendonça de Vasconcellos

DISSERTAÇÃO SUBMETIDA AO CORPO DOCENTE DO INSTITUTO ALBERTO LUIZ COIMBRA DE PÓS-GRADUAÇÃO E PESQUISA DE ENGENHARIA DA UNIVERSIDADE FEDERAL DO RIO DE JANEIRO COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE MESTRE EM CIÊNCIAS EM ENGENHARIA ELÉTRICA.

Orientadores: Marcos Vicente de Brito Moreira
Gustavo da Silva Viana

Aprovada por: Prof. Marcos Vicente de Brito Moreira
Prof. Gustavo da Silva Viana
Prof. João Carlos dos Santos Basilio
Prof. Antonio Eduardo Carrilho da Cunha

RIO DE JANEIRO, RJ – BRASIL
MARÇO DE 2020

Vasconcellos, Augusto Pedro Mendonça de

Redução de Diagnosticadores de Sistemas a Eventos Discretos/Augusto Pedro Mendonça de Vasconcellos. – Rio de Janeiro: UFRJ/COPPE, 2020.

XII, 62 p.: il.; 29, 7cm.

Orientadores: Marcos Vicente de Brito Moreira
Gustavo da Silva Viana

Dissertação (mestrado) – UFRJ/COPPE/Programa de Engenharia Elétrica, 2020.

Referências Bibliográficas: p. 61 – 62.

1. Discrete-event systems. 2. Fault diagnosis. 3. Model reduction. I. Moreira, Marcos Vicente de Brito *et al.* II. Universidade Federal do Rio de Janeiro, COPPE, Programa de Engenharia Elétrica. III. Título.

"You don't get to choose not to
pay a price, you only get to
choose which price you pay."
Jordan B. Peterson

Acknowledgements

First, I thank my mother Nelma de Fátima, whose support in the two most difficult years of my life were fundamental for me to achieve my M.Sc. degree. Therefore, I dedicate this work to you.

I thank my sister Fernanda Mendonça and her husband Erick Palacios for always being willing to help me when necessary.

I thank my father Augusto Allan for continuing to work even in poor health so that I could achieve my goals.

I thank my friends during the postgraduate courses, especially Adalberto Igor and Érika Miranda, for the fun moments during this period.

I would like to thank my great friend Eunápio Lages for the inspiring conversations that made me grow as a person.

A special thank to my co-advisor Gustavo Viana for all corrections in this work, for the theoretical contribution and for clarifying all my ideas.

I deeply thank my advisor Marcos Vicente for his patience and valuable advice. Without your mentoring this work would not be possible.

Resumo da Dissertação apresentada à COPPE/UFRJ como parte dos requisitos necessários para a obtenção do grau de Mestre em Ciências (M.Sc.)

REDUÇÃO DE DIAGNOSTICADORES DE SISTEMAS A EVENTOS DISCRETOS

Augusto Pedro Mendonça de Vasconcellos

Março/2020

Orientadores: Marcos Vicente de Brito Moreira
Gustavo da Silva Viana

Programa: Engenharia Elétrica

Para garantir uma operação confiável em sistemas a eventos discretos, diversos trabalhos na literatura propõem a implementação de diagnosticadores com o objetivo de detectar e isolar eventos de falha não observáveis após a ocorrência de um número limitado de eventos. A eficiência do método de diagnóstico pode ser mensurada pelo atraso para o diagnóstico, definido como o maior número de eventos que ocorreram após a falha até a sua detecção. A principal desvantagem na implementação do diagnosticador proposto na literatura é que seu conjunto de estados pode ser muito grande, exigindo que uma quantidade de memória demasiadamente grande seja implementada em sistemas complexos. Neste trabalho, propomos um algoritmo para computar um diagnosticador reduzido determinístico, que preserve tanto a diagnosticabilidade da linguagem do sistema quanto o mesmo atraso para o diagnóstico que o diagnosticador original. Além disso, mostramos que a estratégia de redução proposta pode levar a um diagnosticador reduzido com menos estados do que utilizamos outras estratégias propostas na literatura.

Abstract of Dissertation presented to COPPE/UFRJ as a partial fulfillment of the requirements for the degree of Master of Science (M.Sc.)

DIAGNOSER REDUCTION OF DISCRETE-EVENT SYSTEMS

Augusto Pedro Mendonça de Vasconcellos

March/2020

Advisors: Marcos Vicente de Brito Moreira
Gustavo da Silva Viana

Department: Electrical Engineering

In order to ensure the reliable operation of Discrete-Event Systems, several works in the literature propose the implementation of diagnosers with the view to detecting and isolating unobservable fault events within a bounded number of event occurrences. The efficiency of the diagnosis method can be measured by the delay of diagnosis, defined as the largest number of event occurrences after the fault until its detection. The main drawback of implementing the diagnoser proposed in the literature is that its state set can be very large, requiring a great amount of memory to be implemented in complex systems. In this work, we propose an algorithm for the computation of a deterministic reduced diagnoser, that preserves the diagnosability of the system language and the same diagnosis delay than the original diagnoser. We show that the proposed reduction strategy can lead to a reduced diagnoser with fewer states than by using other strategies proposed in the literature.

Contents

List of Figures **x**

List of Tables **xii**

1 Introduction **1**

2 Discrete-Event Systems and Fault Diagnosis **4**

2.1 Discrete-Event Systems 4

2.2 Languages 5

2.3 Automata 8

2.3.1 Generated and Marked Languages 10

2.3.2 Operations on Automata 10

2.3.3 Nondeterministic Automata 14

2.3.4 Partially-Observed DES Modeled by Deterministic Automata 15

2.4 Fault Diagnosis 17

2.4.1 Diagnosability of DES 17

2.4.2 Diagnoser Automaton 18

3 Supervisor Reduction for DES **23**

3.1 Controlled DES Under Full Observation 24

3.1.1 Controllability of a Language 26

3.2 Supervisor Reduction Methods 28

3.2.1 Reduction Based on Control Covers 28

3.2.2 Reduction Based on Control Congruence 35

4 Diagnoser Reduction Method **39**

4.1 First Step in Diagnoser Reduction 40

4.2 Procedure of State Merging 40

4.3 Computation of Not Mergeable States 45

4.4 Computation of Deterministic Diagnoser 47

4.5 Computation of Mergeable States Set 49

4.6 Diagnoser Reduction Algorithm 53

5 Conclusions and Future Works **60**

Bibliography **61**

List of Figures

2.1 State transition diagram of automaton G. 9
2.2 Accessible part of G or $Ac(G)$. 11
2.3 Automata G_1 and G_2 of Example 2.6. 13
2.4 Product and parallel compositions between G_1 and G_2 of Example 2.6. 13
2.5 Nondeterministic automaton $\mathcal{G}$ of Example 2.7. 15
2.6 Automaton G of Example 2.8. 17
2.7 $Obs(G)$ of Example 2.8. 17
2.8 Label automaton A_l. 19
2.9 Automaton G of Example 2.9. 20
2.10 Automaton $G_l = G||A_l$ of Example 2.9. 20
2.11 Diagnoser automaton $G_d = Obs(G||A_l)$ of Example 2.9. 20
2.12 Automaton G of Example 2.10. 21
2.13 Automaton $G_l = G||A_l$ of Example 2.10. 22
2.14 Diagnoser automaton $G_d = Obs(G||A_l)$ of Example 2.10. 22

3.1 Feedback supervisory control structure where S/G. 24
3.2 Automaton G of Example 3.1. 27
3.3 Automaton H of Example 3.1. 27
3.4 Automaton G of Example 3.2. 28
3.5 Automaton H of Example 3.2. 28
3.6 Plant G of Example 3.4. 31
3.7 Supervisor S of Example 3.4. 31
3.8 Reduced supervisor S' of Example 3.4. 35
3.9 Reduced supervisor S' of Example 3.5. 38

4.1 Automaton G of Example 4.1. 40
4.2 Automaton G_d of Example 4.1. 41
4.3 Automaton G'_d of Example 4.1. 41
4.4 Automaton G_p. 42
4.5 Automaton G'_p. 42
4.6 Automaton G of Example 4.2. 45

4.7 Automaton G'_d of Example 4.2. 45
4.8 Nondeterministic automaton G''_d of Example 4.2. 45
4.9 Automaton G of Example 4.5. 52
4.10 Automaton G_d of Example 4.5. 52
4.11 Automaton G'_d of Example 4.5. 52
4.12 Merging of states $\{1N\}$ and $\{4N, 7Y\}$ of G'_d of Example 4.6. 56
4.13 Merging of states $\{1N\}$, $\{4N, 7Y\}$ and $\{5N, 8Y\}$ of G'_d of Example 4.6. 57
4.14 Automaton G^r_d of Example 4.6. 58
4.15 Automaton G^r_d of Example 4.6 computed using the method proposed in [1]. 59
4.16 Automaton G^r_d of Example 4.6 computed using the method proposed in [2]. 59

List of Tables

3.1 Disabled and ineligible state sets for all states of S of Example 3.4. . . 31

3.2 Compatible states of Example 3.4. 33

3.3 Mergeable states of Example 3.4. 34

Chapter 1

Introduction

In order to ensure the reliable operation of industrial automation systems, it is necessary to efficiently diagnose the occurrence of faults, *i.e.*, to detect and isolate the fault that has occurred before it damages the system components or cause risk to operators. If the system can be abstracted as a Discrete-Event System (DES), that is, its state set is discrete and the transition between states is event-driven, then the fault can be modeled as an event that must be identified by the diagnosis system. In this case, the delay of diagnosis, defined as the largest number of event occurrences that the system may execute after the fault until its detection [3–5], can be used to determine the efficiency of the diagnosis method.

In [6], a diagnoser is constructed to detect and isolate unobservable fault events, and the property of diagnosability, which is related with the capability of identifying the fault occurrence within a bounded number of event occurrences, is introduced. Diagnosers can be used to verify the diagnosability of the system and for online diagnosis. The main drawback of implementing directly the diagnoser proposed in [6] is that, since it is constructed based on an observer, then its state set grows, in the worst-case, exponentially with the number of states of the system model. Although, in [7], it is shown that in the average-case the diagnoser grows polynomially with the number of system states, *i.e.*, it can still have a large number of states, requiring a large amount of memory to be implemented in complex systems. Thus, it is desirable to reduce the size of the diagnoser for implementation, preserving the diagnosability of the system and the delay for diagnosis. In order to do so, it is necessary that the language of the reduced diagnoser simulates the original language of the diagnoser. In general, however, the reduction of states of an automaton leads to a larger language in comparison with the original one. This is due to the generation of an exceeding language by the introduction of new cycles in the reduced automaton.

Over the years, the deterministic finite automaton (DFA) minimization problem has been addressed for several purposes, as presented in [8], [2], [9] and [10]. In [9],

the minimization of a DFA is based on finding the states of the automaton that are equivalent in the sense that they can be merged without changing the marked language of the system. Concerning the diagnoser reduction problem, one of the most relevant works in the literature is [8], which presents a state-based approach for online fault diagnosis.

Other reduction approaches have been addressed in the literature, most notably, in the supervisor control theory [1, 2, 11]. In [1], the reduction process is based on finding *control covers*, that are sets formed of sets of states that behave consistently regarding their control actions, *i.e.*, for any pair of states of a subset of a control cover, all events that are enabled by one of the states cannot be disabled by the other. The limitation of the method proposed in [1] is that, in some cases, the determinization of the automaton, or even a reduction of the supervisor, cannot be guaranteed. To overcome this problem, [2] introduced the concept of *control congruence*, which is a special case of control cover. If the subsets that form the control cover are pairwise disjoint, then the control cover is called a control congruence. The determinism of the reduced supervisor is guaranteed using the control congruence, but not the computation of the minimal supervisor, which is shown in [2] to be an NP-hard problem.

Recently, in [10], a procedure is introduced to reduce the number of states of an automaton while preserving some properties of interest. Their approach is based on finding a *reduction cover*, which is an extension of the concept of control cover [12]. In order to do this, the algorithm proposed in [2] is adapted to compute a reduction partition that is necessary to compute the reduction cover. The algorithms presented in [2] and [10] for the computation of covers are dependent on an ordering of the states of the automaton that must be reduced. However, this ordering of states is not exploited in [2] and [10] to find a reduced automaton with a smaller number of states.

In this work, we propose an algorithm for the computation of a deterministic reduced diagnoser, that preserves the diagnosability of the system language, and that has the same diagnosis delay than the original diagnoser. As in [2] and [10], the sets of merged states of the reduced diagnoser are disjoint. At each step of the algorithm for computing the reduced diagnoser, states are selected to be merged according to a criterion that takes into account the possibility of merging other states of the diagnoser in the next step of the reduction procedure. In other words, differently from the other methods presented in the literature, the proposed method does not depend on the ordering of states previously given as an input of the algorithm. An example is used throughout the text to illustrate the application of the method, and to show that the proposed reduction strategy can lead to a reduced diagnoser with fewer states than the diagnoser obtained by considering the same criterion used

in [2] and [10], while maintaining the diagnoser deterministic, differently from the method proposed in [1] to reduce supervisors.

The present work is organized as follows. Chapter 2 is a brief review over the theory of discrete-event systems (DES), that are necessary to full understanding of the concept of fault diagnosis. Also in Chapter 2, necessary and sufficient conditions for fault diagnosis using the diagnoser automaton are presented. In Chapter 3 are presented the basic fundamentals of supervisory control theory. In Chapter 4 is developed the main scope of this work, where the method of diagnoser reduction is presented. Finally, in Chapter 5, we draw conclusions and suggestions of future works.

Chapter 2

Discrete-Event Systems and Fault Diagnosis

In this chapter, we present the necessary background on Discrete Event Systems (DES), and on failure diagnosis of DES. The theoretical foundations of DES presented in this chapter are based on [13]. The structure of the chapter is as follows. In Section 2.1, the main formalisms for DES are presented. In Section 2.2, we present the concept of a given language for a DES. In Section 2.3, the automata theory is described. Finally, the main concepts associated with fault diagnosis are presented in Section 2.4.

2.1 Discrete-Event Systems

As presented in [13], discrete-event systems (DES) are dynamical systems, whose state-spaces are discrete sets that normally depend of the occurrency of asynchronous discrete-events over time. Starting with this definition, we can immediately identificate two important characteristics of DES: they have discrete state variables and they are driven by events.

The state is described by a discrete set, for instance: numerical values belonging to the set $\mathbb{N}$ or $\mathbb{Z}$ ($\{0, 1, 2, ...\}$), symbolical values ($\{open, closed, stuck\}$). On the other hand, the events can be defined by instantaneous occurrences that make transitions on the states of the system. In other words, the events can be related to a specific action (e.g. press a button, activate a switch), a spontaneous occurrence (e.g. break of a mechanical piece, an alarm activation, the turning off of a heater), or the result of conditions already predicted (e.g. the level of a tank getting on a determined value).

The system behavior in the DES framework is driven by sequences of events. All possible sequences of the events that can be generated by a given DES describe

the language of this system, which is defined over a set of events (alphabet) of the system. A review over the concept of a given language will be made in the next section to clarify this idea.

2.2 Languages

Before we introduce the definition of a language, some remarks need to be made. The first one is the set of events of a DES, represented by Σ, is an alphabet supposed to be finite.

The behavior of a DES can be described in terms of event sequences. Those sequences are interpreted like "words" of a language. Thus, the language generated by a given DES is the set of all of the event sequences with finite length, or words, generated by the system. Then, since we have the knowledge of the language generated by the system and its initial state, we can describe the future behavior of the system. With that in mind, we can say that the language is kind of a mathematical formalism that models and describes the behavior of a DES. Next, we formally the concept of language.

Definition 2.1 *(Language) A language defined over an event set Σ is a set of sequences with finite length formed of events in Σ.*

In order to make important definitions and examples, some symbols must be presented. The symbol σ will be used to represent a generic event, as well as $\emptyset$ will be the empty language and ε the empty sequence. For a sequence s defined over Σ, its length can be represented by $|s|$. By definition, $|\varepsilon| = 0$. In addition, $\sigma \in s$ means that event σ belongs to a sequence s.

The language of a DES belongs to the set of all the sequences of finite length that are formed by elements of Σ, including the empty sequence ε. This set is called the Kleene-closure of Σ, denoted as Σ^*. In particular, $\emptyset$, Σ and Σ^* are considered event sets. Moreover, we can say that 2^Σ is the power set of Σ or, in other words, the set that contains all the subsets of Σ.

Example 2.1 *Let $\Sigma = \{a, b, c\}$ be a set of events. For instance, we may define the languages:*

- $L_1 = \{\varepsilon, a, b, c, ab, ac, bc, abc\}$*;*
- $L_2 = \{a, bb, abbc, aaabcc\}$*;*
- $L_3 = \{$*the set of sequences that contains* $ab\}$*.*

The usual set operations, such as union, intersection, difference and complement with respect to Σ^*, are applicable to languages since languages are sets. In additon, there are other operations, like *concatenation* and *Kleene-closure*, which are formally defined as follows.

Definition 2.2 *(Concatenation) Let $L_a, L_b \subseteq \Sigma^*$. Then, the concatenation $L_a L_b$ is defined as:*

$$L_a L_b = \{s \in \Sigma^* : (s = s_a s_b), (s_a \in L_a) \text{ and } (s_b \in L_b)\} .$$

A sequence s is in $L_a L_b$ if it is formed by the concatenation of a sequence $s_a \in L_a$ and $s_b \in L_b$.

Definition 2.3 *(Kleene-closure) Consider $L \subseteq \Sigma^*$. Then, the Kleene-closure of L, denoted by L^*, can be defined as:*

$$L^* = \{\varepsilon\} \cup L \cup LL \cup LLL \cup LLLL \ldots$$

The elements of L^* are composed by the concatenation of the elements of L, including the empty sequence ε. The Kleene-closure is idempotent, namely, $(L^*)^* = L^*$.

Definition 2.4 *(Prefix-closure) The prefix-closure of a language L consists of all prefixes of all traces in L. A sequence $t \in \Sigma^*$ is prefix of a sequence $s \in \Sigma^*$ if there exists a sequence $v \in \Sigma^*$ such that $tv = s$. Then, both s and ε are prefixes of s. The prefix-closure of L, denoted as $\overline{L}$, can be formally defined as:*

$$\overline{L} = \{s \in \Sigma^* : (\exists t \in \Sigma^*)[st \in L]\} .$$

L is said to be prefix-closed if $L = \overline{L}$. Therefore, language L is prefix-closed if all prefixes of every sequence in L are also an element of L.

Example 2.2 *Consider $\Sigma = \{a, b, c\}$ and languages $L_1 = \{\varepsilon, a, aab\}$ and $L_2 = \{c\}$ defined over Σ. Note that L_1 and L_2 are not prefixed-closed, since $aa \notin L_1$ and $\varepsilon \notin L_2$. Thus, the prefix-closures of L_1 and L_2 are, respectively, $\overline{L}_1 = \{\varepsilon, a, aa, aab\}$ and $\overline{L}_2 = \{\varepsilon, c\}$.*

Definition 2.5 *(Post-language) Consider $L \subseteq \Sigma^*$. Then, the post-language of L after s, denoted by L/s, is defined as:*

$$L/s = \{t \in \Sigma^* : st \in L\} .$$

This definition allows us to conclude that $L/s = \emptyset$ if $s \notin \overline{L}$.

Another type of operation perfomed on sequences and languages is the natural projection, or simply projection, denoted by P, and is defined as follows.

Definition 2.6 *(Projection) Let Σ_s and Σ_l be event sets such that $\Sigma_s \subset \Sigma_l$ and σ any event. Then, we can define the projection of a sequence as:*

$$P : \Sigma_l^* \to \Sigma_s^*,$$

with the following properties:

(i) $P(\varepsilon) := \varepsilon$;

(ii) $P(\sigma) := \begin{cases} \sigma, & \text{if} \quad \sigma \in \Sigma_s \\ \varepsilon, & \text{if} \quad \sigma \in \Sigma_l \setminus \Sigma_s \end{cases}$;

(iii) $P(s\sigma) := P(s)P(\sigma)$ *for* $s \in \Sigma_l^*$, $\sigma \in \Sigma_l$.

According to Definition 2.6, the projection operation takes a sequence and 'erases' events on Σ_l that does not belong to Σ_s.

The projection P and its inverse P^{-1} can be extended to languages simply applying them to all the event sequences on that language.

Thus, for $L \subseteq \Sigma_l^*$, we have:

$$P(L) := \{t \in \Sigma_s^* : (\exists s \in L)[P(s) = t]\}.$$

For $L_s \subseteq \Sigma_s^*$, we can define its inverse as:

$$P^{-1}(L_s) := \{s \in \Sigma_l^* : (\exists t \in L_s)[P(s) = t]\}.$$

In order to illustrate the concepts of this subsection, consider the following example.

Example 2.3 *Let us consider again the set of events $\Sigma = \{a, b, c\}$, and languages $L_1 = \{\varepsilon, a, aab\}$ and $L_2 = \{\varepsilon, c\}$. Since $\overline{L}_1 = \{\varepsilon, a, aa, aab\}$ and $\overline{L}_2 = \{\varepsilon, c\}$, then $L_1 \neq \overline{L}_1$ and $L_2 = \overline{L}_2$. Consequently, L_1 is not prefix-closed, but L_2 is prefix-closed. In addition, note that:*

$$\begin{aligned} L_1^* &= \{\varepsilon, a, aab, aa, aaab, aaba, \ldots\} \\ L_2^* &= \{\varepsilon, c, cc, ccc, \ldots\} \\ L_1L_2 &= \{\varepsilon, a, aab, c, ac, aac, aabc\} \\ L_1/a &= \{\varepsilon, ab\} \end{aligned}$$

If we now define projection $P : \Sigma^* \rightarrow \Sigma_s^*$ such that $\Sigma_s = \{b, c\}$, then:

$$\begin{aligned}
P(abc) &= \{bc\} \\
P^{-1}(\varepsilon) &= \{a\}^* \\
P^{-1}(bc) &= \{a\}^*\{b\}\{a\}^*\{c\}\{a\}^* \\
P(L_1) &= \{\varepsilon, b\} \\
P^{-1}(L_2) &= \{\{a\}^*, \{a\}^*\{c\}\{a\}^*\}
\end{aligned}$$

To some practical issues, the use of languages can be considerably complex. In this work, we use automata as framework for representing and manipulating languages. For this reason, next section will present a brief review of automata theory.

2.3 Automata

An automaton is a device that is capable of representing a language according to well-defined rules [13]. In the following, we formally define a deterministic automaton.

Definition 2.7 *(Deterministic automata) A deterministic automata, denoted by G, is a five-tuple*

$$G = (X, \Sigma, f, x_0, X_m),$$

where:

- *X is the set of states of the automata;*
- *Σ is the finite set of events associated with G;*
- *$f : X \times \Sigma^* \rightarrow X$ is the state transition function of the states;*
- *x_0 is the initial state of the system;*
- *$X_m \subseteq X$ is the set of marked states;*

The transition function $f(x, \sigma) = y$ means that there is a transition rotulated by event σ from state x to state y; generally, f is a partial function in its own domain. Let $\Gamma_G : X \rightarrow 2^\Sigma$ be the feasible event function, where $\Gamma_G(x)$ is the set of events that are feasible in state $x \in X$.

The dynamic behavior represented by automaton G works as follows. It starts in the initial state x_0 and due to an ocurrence of an event $e \in \Gamma(x_0) \subseteq \Sigma$, the automaton makes the transition to the state $f(x_0, \sigma) \in X$. The process continues based on the transitions which f is defined.

A common way to represent an automaton is through the state transition diagram. The states are represented by circles, marked states by two concentric circles and the transitions by oriented arcs. States are marked when it is necessary to attach a special meaning to them, for instance, the end of a task. The next example illustrates the state transition diagram.

Example 2.4 *Let G be an automaton which its state transition diagram is depicted in Figure 2.1. Based on the diagram, we have:*

- $X = \{0, 1, 2, 3, 4, 5, 6\}$*;*
- $\Sigma = \{a, b, c\}$*;*
- *The state transition function of G is given by $f(0, a) = 1$, $f(1, b) = 3$, $f(1, c) = 2$, $f(3, a) = 3$, $f(3, b) = 4$, $f(4, a) = 5$, $f(5, c) = 4$ and $f(6, a) = 4$;*
- *The feasible event sets of each state are given by $\Gamma_G(0) = \{a\}$, $\Gamma_G(1) = \{c, b\}$, $\Gamma_G(2) = \emptyset$, $\Gamma_G(3) = \{a, b\}$, $\Gamma_G(4) = \{a\}$, $\Gamma_G(5) = \{c\}$ and $\Gamma_G(6) = \{a\}$;*
- *The initial state of G is $x_0 = 0$;*
- *The set of marked states of G is $X_m = \{2\}$.*

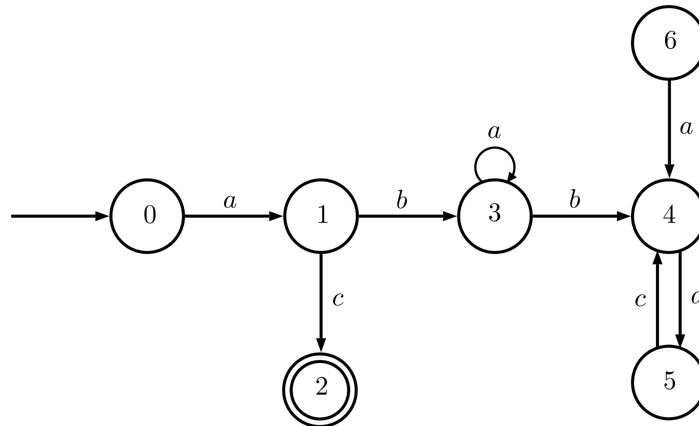


Figure 2.1: State transition diagram of automaton G.

From Example 2.4, it is clear that automata can represent a subset of sequences of Σ^* or, in other words, a language over the alphabet Σ. Thus, we can define two languages: the generated and marked languages. These notions will be presented in next section.

2.3.1 Generated and Marked Languages

The language generated by an automaton G, denoted by $L(G)$, is the set formed by all the event sequences that can be defined starting in the initial state. The language marked by G, denoted by $L_m(G)$, contains all event sequences that takes the initial state of the automaton to a marked state. We can define the generated and the marked languages as follows.

Definition 2.8 *(Generated language) Since f is a partial function, then the language generated by G, denoted as $L(G) = L$, can be defined as*

$$L = \{s \in \Sigma^* : f(x_0, s)!\},$$

where $f(x_0, s)!$ means that $f(x_0, s)$ is defined.

Note that $L(G)$ is, by definition, prefix-closed, since a path is only possible if all the correspondent prefixes are also possibles. Furthermore, it is possible that the events defined in Σ do not belong to the state transition diagram of G and, therefore, do not belong to $L(G)$ as well.

Definition 2.9 *(Marked language) The language marked by $G = (X, \Sigma, f, x_0, X_m)$, denoted by $L_m(G)$ is defined as*

$$L_m(G) := \{s \in L(G) : f(x_0, s) \in X_m\}.$$

Note that $L_m(G)$ will always be a subset of $L(G)$, since $L_m(G)$ is composed by all the s sequences such that $f(x_0, s) \in X_m$. As $L_m(G)$ is not necessarily prefix-closed, not all the states of X need to be marked. When an automaton does not have states, it is said that it generates and marks the empty set.

In the next section, we present some operations that can be applied to automata.

2.3.2 Operations on Automata

In order to analyze DES modeled by automata we first need to define the set of operations capable of modifying properly the state transition diagram set of an automaton.

Next, we present an operation that change a single automaton, called unary operation: accessibility or accessible part. It will also be presented the product and the parallel compositions, which are fundamental to obtain the fault diagnosers.

Accessible Part

With the definitions of $L(G)$ and $L_m(G)$, one may notice that we can erase all the states of G that are not accessible or reachable, starting from x_0, by some sequence in $L(G)$, without affecting not only the generated language of G, but also its marked language. When we remove a state, we also remove all transitions linked to that state.

Definition 2.10 *(Accessible part) Let $G = (X, \Sigma, f, x_0, X_m)$ be an automaton. The accessible part or accessibility of G, denoted by $Ac(G)$, is defined as*

$$Ac(G) = (X_{ac}, \Sigma, f_{ac}, x_0, X_{ac,m}),$$

where:

(i) $X_{ac} = \{x \in X : (\exists s \in \Sigma^*)[f(x_0, s) = x]\}$;

(ii) $X_{ac,m} = X_m \cap X_{ac}$;

(iii) $f_{ac} = f \mid_{X_{ac} \times \Sigma \to X_{ac}}$.

The item *(iii)* means that we are restricting f to the smaller domain of the accessible states X_{ac}. Note that the event set of $Ac(G)$ remains equal to G, even if some event of the set does not be in the states transition diagram of $Ac(G)$. Thus, the operation Ac does not changes $L(G)$ and $L_m(G)$.

Example 2.5 *Let us consider again automaton G depicted in Figure 2.1. Note that state 6 is the only state not accessible from initial state 0. Thus, we must remove state 6 and all its transitions to obtain $Ac(G)$, depicted in Figure 2.2.*

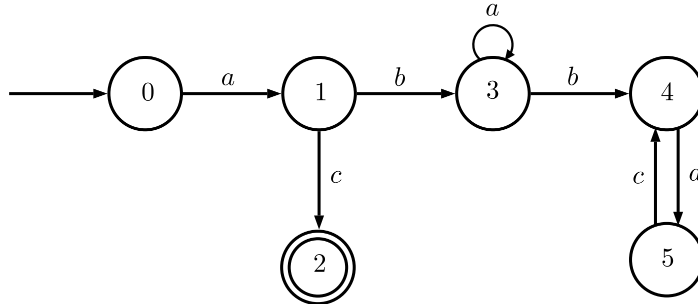


Figure 2.2: Accessible part of G or $Ac(G)$.

Product Composition

The product composition, denoted by $\times$, is a composition of two automata that allows the occurrence of events that are common to both. The following definition describe this operation mathematically [13].

Definition 2.11 *(Product composition) Consider the automata*

$$\begin{aligned} G_1 &= (X_1, \Sigma_1, f_1, x_{0,1}, X_{m_1}) \\ G_2 &= (X_2, \Sigma_2, f_2, x_{0,2}, X_{m_2}). \end{aligned}$$

The product composition between G_1 and G_2 will be given by the automaton

$$G_1 \times G_2 := Ac(X_1 \times X_2, \Sigma_1 \times \Sigma_2, f, (x_{0,1}, x_{0,2}), X_{m_1} \times X_{m_2}),$$

where:

(i) $f((x_1, x_2), \sigma) := \begin{cases} (f_1(x_1, \sigma), f_2(x_2, \sigma)), & \text{if } \sigma \in \Gamma_1(x_1) \cap \Gamma_2(x_2) \\ \text{undefined}, & \text{otherwise} \end{cases};$

(ii) $\Gamma_{1\times 2}(x_1, x_2) = \Gamma_1(x_1) \cap \Gamma_2(x_2)$ *is the feasible event function of $G_1 \times G_2$.*

Considering the right side of the definition of $G_1 \times G_2$, it can be noted that we are only interested in the accessible part of the automaton. In other words, an event only occurs in $G_1 \times G_2$ if it occurs in both G_1 and G_2. In the product operation, the transitions of two automata should always be synchronized in a common event. Then, we can say that both generated and marked languages by the product $G_1 \times G_2$ can be given by:

$$\begin{aligned} L(G_1 \times G_2) &= L(G_1) \cap L(G_2), \\ L_m(G_1 \times G_2) &= L_m(G_1) \cap L_m(G_2). \end{aligned}$$

Parallel Composition

As stated in [13], the more commonly used method in building complete models of systems starting from individual components is made by parallel composition (or synchronous) of automata, where each automaton represents a local component (or subsystem) of the global system. Therefore, the definition of parallel composition is made as follows.

Definition 2.12 *(Parallel composition) Let G_1 and G_2 be two automata. Then, the parallel composition between $G_1 = (X_1, \Sigma_1, f_1, x_{0,1}, X_{m_1})$ and $G_2 = (X_2, \Sigma_2, f_2, x_{0,2}, X_{m_2})$, denoted by $G_1 \| G_2$, is defined as:*

$$G_1 \| G_2 = Ac(X_1 \times X_2, \Sigma_1 \cup \Sigma_2, f_{1\|2}, (x_{0,1}, x_{0,2}), X_{m_1} \times X_{m_2}),$$

where $f_{1\|2}((x_1,x_2),\sigma) = (f_1(x_1,\sigma), f_2(x_2,\sigma))$, *if* $\sigma \in \Gamma_{G_1}(x_1) \cap \Gamma_{G_2}(x_2)$, *and* $\Gamma_{G_i} : X_i \to 2^{\Sigma_i}$, $i = 1,2$, $f_{1\|2}((x_1,x_2),\sigma) = (f_1(x_1,\sigma), x_2)$, *if* $\sigma \in \Gamma_{G_1}(x_1)\backslash\Sigma_2$, $f_{1\|2}((x_1,x_2),\sigma) = (x_1, f_2(x_2,\sigma))$, *if* $\sigma \in \Gamma_{G_2}(x_2)\backslash\Sigma_1$, *or, undefined, otherwise.*

It can be noted that a common event from the automata G_1 and G_2 can only occur when both are in states whose active event sets have this event as an element. Private events (the ones belonging to $\Sigma_1 \setminus \Sigma_2$ or to $\Sigma_2 \setminus \Sigma_1$) are not subject to limitations, being able to execution as long as possible. Therefore, the parallel composition synchronizes only common events to both G_1 and G_2.

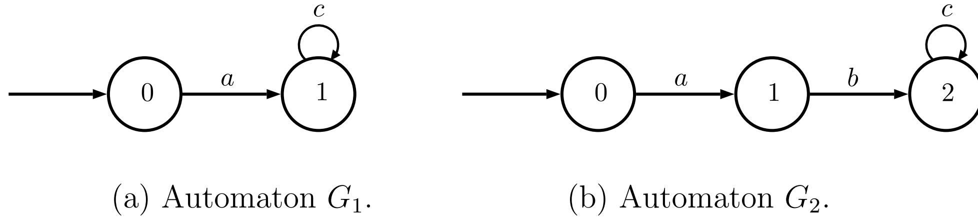
In order to define the generated and marked languages of automaton $G_1\|G_2$, it is possible to use Definition 2.6, rewriting it as follows:

$$P_i : (\Sigma_1 \cup \Sigma_2)^* \to \Sigma_i^* \text{ for } i = 1,2.$$

Considering the use of projections, we now can obtain the resulting languages of the parallel composition between G_1 and G_2:

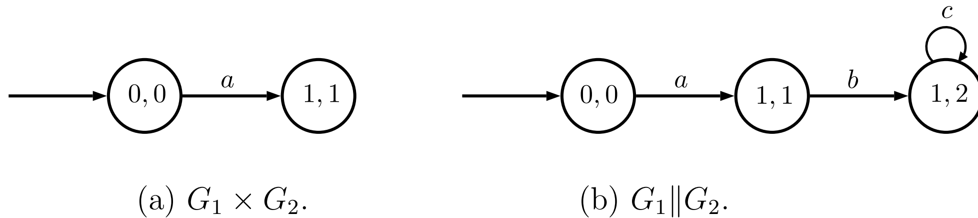
$$\begin{aligned} L(G_1\|G_2) &= P_1^{-1}[L(G_1)] \cap P_2^{-1}[L(G_2)], \\ L_m(G_1\|G_2) &= P_1^{-1}[L_m(G_1)] \cap P_2^{-1}[L_m(G_2)]. \end{aligned}$$

Example 2.6 *Let G_1 and G_2 be automata depicted in Figures 2.3(a) and 2.3(b), respectively, where $\Sigma_1 = \{a, c\}$ and $\Sigma_2 = \{a, b, c\}$. The product and parallel compositions between those automata are depicted in Figures 2.4(a) and 2.4(b), respectively.*



(a) Automaton G_1. (b) Automaton G_2.

Figure 2.3: Automata G_1 and G_2 of Example 2.6.



(a) $G_1 \times G_2$. (b) $G_1\|G_2$.

Figure 2.4: Product and parallel compositions between G_1 and G_2 of Example 2.6.

2.3.3 Nondeterministic Automata

A nondeterministic automaton, denoted by $\mathcal{G}$, is a five-tuple $\mathcal{G} = (X, \Sigma \cup \{\varepsilon\}, f_{nd}, x_0, X_m)$, where the elements of $\mathcal{G}$ have the same interpretation as in the deterministic automaton G, with the exception that the transition function can be nondeterministic,

$$f_{nd} : X \times \Sigma \cup \{\varepsilon\} \to 2^X,$$

and the initial state can be defined as a set $x_0 \subseteq X$.

In order to define the language generated by $\mathcal{G}$, it is necessary to extend the domain of f_{nd} to $X \times \Sigma^*$, obtaining the extended transition function f_{nd}^e. Let $\varepsilon R(x)$ denote the ε-reach of a state x, *i.e.*, the set of states reached from x by following transitions labeled with ε, including state x. The ε-reach can be extended to a set of states $B \subseteq X$ as

$$\varepsilon R(B) = \cup_{x \in B} \varepsilon R(x).$$

The extended nondeterministic transition function $f_{nd}^e : X \times \Sigma^* \to 2^X$, can be defined recursively as $f_{nd}^e(x, \varepsilon) = \varepsilon R(x)$, and $f_{nd}^e(x, s\sigma) = \varepsilon R[\{z : z \in f_{nd}(y, \sigma) \text{ for some state } y \in f_{nd}^e(x, s)\}]$. Thus, the language generated by $\mathcal{G}$ can be defined as

$$L(\mathcal{G}) = \{s \in \Sigma^* : (\exists x \in x_0)[f_{nd}^e(x, s) \text{ is defined}]\}.$$

And the language marked by $\mathcal{G}$ can be defined as

$$L_m(\mathcal{G}) = \{s \in \Sigma^* : (\exists x \in x_0)[f_{nd}^e(x, s) \cap X_m \neq \emptyset]\}.$$

In this work, the marked language and transitions labeled with ε will not be necessary when computing nondeterministic automata. Therefore, they will be omitted.

In order to illustrate the nondeterministic automaton, consider the following example.

Example 2.7 *Consider the nondeterministic automaton, $\mathcal{G}$, depicted in Figure 2.5, where $x_0 = \{0\}$. Note that the transition function assumes values in 2^X, for $x \in X$, for instance, $f_{nd}(1, b) = \{2, 3\}$. This configuration suggests uncertainty in the dynamic evolution of the system, considering that, when the system is in state* 1 *and event b occurs, it is not possible to be sure if the system has moved either to state* 2 *or* 3.

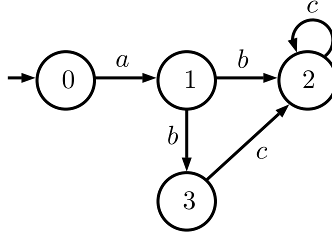


Figure 2.5: Nondeterministic automaton $\mathcal{G}$ of Example 2.7.

2.3.4 Partially-Observed DES Modeled by Deterministic Automata

When exist some events in a given DES whose occurrence cannot be seen by an outside observer, it is said that the system is a partially-observed DES. It is said that an event is observable if its occurrence can be observed by an external agent (like a sensor) and communicated to the system observer. Otherwise, the event is unobservable. The cause of an unobservation can be due to the absence of a sensor to record the occurrence of the event, or to the fact that the event occurred at a remote location, where its occurrence could not be verified. Therefore, it is necessary to make a state estimation when the analysis of the behavior of the system is concerned.

To this end, we can make the following suppositions:

- $\Sigma = \Sigma_o \dot{\cup} \Sigma_{uo}$ is the set of all events of a partially-observed DES;
- Σ_o is the set of observable events of the system;
- Σ_{uo} is the set of unobservable events of the system;

The representation of a unobservable event in a given state transition diagram of an automaton is made with a dashed line on the correspondent oriented arc. Unless indicated otherwise in the automaton, the event will be considered observable. The corresponding automaton G that models the partially-observed DES will be deterministic, and if it has unobservable events, it is said that G is a partially-observed automaton.

In order to define the observer automaton of G, denoted by $Obs(G)$, we need first to introduce the unobservable reach of a state. The unobservable reach of a state x, $UR(x)$, generates the set of states that are reachable from x after the execution of a sequence of unobservable events in Σ_{uo}. Thus,

$$UR(x) = \{y \in X : (\exists t \in \Sigma_{uo}^*) \, [(f(x,t) = y)]\} \, .$$

The unobservable reach can be extended to a set os states $A \in 2^X$ as:

$$UR(A) = \bigcup_{x \in A} UR(x).$$

Then, the observer of G, $Obs(G)$, can be defined as follows:

$$Obs(G) = (X_{obs}, \Sigma_o, f_{obs}, x_{0_{obs}}, X_{m_{obs}}),$$

where $X_{obs} \subseteq 2^X$, $x_{0_{obs}} = UR(x_0)$, $X_{m_{obs}} = \{x_{obs} \in X_{obs} : x_{obs} \cap X_m \neq \emptyset\}$, and for all $x_{obs} \in X_{obs}$, $\Gamma_{G_{obs}}(x_{obs}) = \bigcup_{x \in x_{obs}} \Gamma_G(x) \cap \Sigma_o$, $f_{obs}(x_{obs}, \sigma) = \bigcup_{(x \in x_{obs}) \wedge (f(x,\sigma)!)} UR(f(x, \sigma))$, if $\sigma \in \Gamma_{G_{obs}}(x_{obs})$, or undefined, otherwise. Considering that diagnosers do not have marked states, X_m and $X_{m_{obs}}$ will be omitted in this work when computing $Obs(G)$. The following algorithm computes $Obs(G)$ [13].

Algorithm 2.1: Construction of automaton $Obs(G)$

Input: Automaton $G = (X, \Sigma, f, x_0)$, and set Σ_o
Output: Automaton $Obs(G) = (X_{obs}, \Sigma_o, f_{obs}, x_{0_{obs}})$

1 Set $\Sigma_{uo} = \Sigma \setminus \Sigma_o$
2 Define $x_{0_{obs}} = UR(x_0)$
3 Set $X_{obs} = \{x_{0_{obs}}\}$
4 Set $\tilde{X}_{obs} = X_{obs}$
5 $\hat{X}_{obs} \leftarrow \tilde{X}_{obs}$
6 $\tilde{X}_{obs} \leftarrow \emptyset$
7 **for** $A \in \hat{X}_{obs}$ **do**
8 $\quad \Gamma_{G_{obs}}(A) = \left(\bigcup_{x \in A} \Gamma(x)\right) \cap \Sigma_o$
9 $\quad$ **for** $\sigma \in \Gamma_{G_{obs}}(A)$ **do**
10 $\quad\quad f_{obs}(A, \sigma) = UR(\{x \in X : (\forall y \in B)[x = f(y, \sigma)]\})$
11 $\quad\quad \tilde{X}_{obs} \leftarrow \tilde{X}_{obs} \cup f_{obs}(A, \sigma)$
12 $X_{obs} \leftarrow X_{obs} \cup \tilde{X}_{obs}$
13 Repeat Steps 5 to 12 until the entire accessible part of $Obs(G)$ has been obtained

---

All sequences of $L(G)$ that does not include unobservable events form the generated language observed by automaton G. This language can be obtained through the projection operation, P_o, where $P_o : \Sigma^* \rightarrow \Sigma_o^*$. Thus, $L(Obs(G)) = P_o[L(G)]$. The following example illustrates the construction of observers.

Example 2.8 *Let G be the automaton depicted in Figure 2.6 and suppose that event d is unobservable, i.e., $\Sigma = \{a, b, c, d\}$, where $\Sigma_o = \{a, b, c\}$ and $\Sigma_{uo} = \{d\}$. Then, according to Algorithm 2.1, we can construct the observer of G, $Obs(G)$, depicted in Figure 2.7. When event a occurs, the system moves to state 1, but it can reach state 3 through unobservable transition d, leading $Obs(G)$ to reach state $\{1, 3\}$. When*

event b occurs in state $\{1,3\}$, the system moves to state 2, considering that both states, 1 and 3, reaches state 2 through event b. Then, if event a occurs in state 2, the system moves to state 1, but it can also reach state 3 again considering that d is an unobservable event. Finally, if event c occurs in state 2, the system remains at the state.

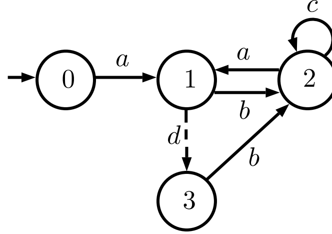


Figure 2.6: Automaton G of Example 2.8.

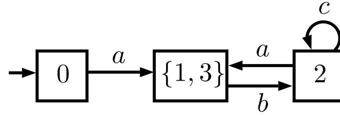


Figure 2.7: $Obs(G)$ of Example 2.8.

In the next section we present the formal definition of fault diagnosis of a DES, and the conditions for diagnosability of a language.

2.4 Fault Diagnosis

The main objective in the process of detection and fault diagnosis is to identify the cause of poor system functioning. In order to diagnose faults in a system, we first need to define the concept of diagnosability of a DES, which is presented in the following section.

2.4.1 Diagnosability of DES

In [6], the property of the system language related with the capability of diagnosing the occurrence of the fault event σ_f, called *diagnosability*, is presented. To define formally this property, in [6] the following assumptions are considered: (i) language L is live, *i.e.*, $\Gamma(x_i) \neq \emptyset$ for all $x_i \in X$, and (ii) the plant automaton G does not contain cyclic paths formed with unobservable events only. In this work, we make the same assumptions presented in [6].

Before we do the proper definition of the concept of diagnosability, it is necessary to define the fault-free behavior of G with respect to $\Sigma_f = \{\sigma_f\}$, which is presented as follows.

Definition 2.13 *(Fault-free behavior) Let $L(G) = L$ be the language generated by automaton G and L_N the prefix-closed language formed by all the sequences of L that does not have any fault event from the set Σ_f. Then, the fault-free behavior of the system given by G, with respect to $\Sigma_f = \{\sigma_f\}$, will be modeled by subautomaton of G, G_N, that generates language L_N.*

Then, the formal definition of the diagnosability of L can be stated as follows [6].

Definition 2.14 *(Diagnosability) Let L be the live and prefix-closed language generated by the system, and $L_N \subset L$ be the fault-free language of L. Let $P_o : \Sigma^* \rightarrow \Sigma_o^*$ be a projection operation. Then, L is said to be diagnosable with respect to projection P_o and Σ_f, if*

$$(\exists z \in \mathbb{N})\,(\forall s \in L \setminus L_N)\,(\forall st \in L \setminus L_N, |t| \geq z)$$

$$\Rightarrow \left(\forall \omega \in P_o^{-1}\left(P_o(st)\right) \cap L, \omega \in L \setminus L_N\right).$$

According to Definition 2.14, L is not diagnosable with respect to P_o and Σ_f if, and only if, there exists an arbitrarily long length faulty sequence st with the same observation than a fault-free sequence ω in L_N, *i.e.*, $P_o(st) = P_o(\omega)$.

If L is diagnosable with respect to P_o and Σ_f, then it is possible to define the delay bound for diagnosis, denoted as z^*, as the minimum value of z that satisfies the diagnosability condition of Definition 2.14.

In the following section, we define the diagnoser automaton.

2.4.2 Diagnoser Automaton

In [6], a diagnoser automaton is proposed to perform online diagnosis and to verify the system diagnosability. In order to do so, it is first defined a label automaton $A_l = (X_l, \Sigma_f, f_l, x_{0,l})$, depicted in Figure 2.8, where $X_l = \{N, Y\}\,, x_{0,l} = \{N\}\,, f_l(N, \sigma_f) = Y$, and $f_l(Y, \sigma_f) = Y$. The label Y indicates the occurrence of the fault event σ_f, and the label N means that the fault has not occurred.

Then, the diagnoser automaton is given by

$$G_d = Obs(G_l) = (X_d, \Sigma_o, f_d, x_{0_d}), \tag{2.1}$$

where $G_l = G \| A_l$. From Equation 2.1, it can be noted that $L(G_d) = P_o(L(G \| A_l)) = P_o(L(G))$, and Σ_o is the set of observable events of G.

Note that the states of G_d have the following form $x_d = \{(x_1, l_1), \ldots, (x_n, l_n)\}$, where $x_i \in X$ and $l_i \in \{Y, N\}$, for $i = 1, \ldots, n$. If label $l_i = Y$ for $i = 1, \ldots, n$, then x_d is said to be a *positive* state. On the other hand, if $l_i = N$ for $i = 1, \ldots, n$, x_d is said to be a *negative* state. Finally, if there exists $l_i = Y$ and $l_j = N$, $i \neq j$, x_d is said to be an *uncertain* state.

The set of states of the diagnoser automaton can be partitioned as $X_d = X_Y \dot{\cup} X_N \dot{\cup} X_{NY}$, where X_Y is the set formed of all positive states of G_d, X_N is the set formed of all negative states of G_d, and X_{NY} is the set formed of all uncertain states of G_d.

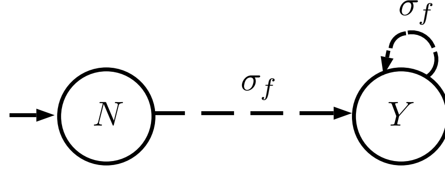


Figure 2.8: Label automaton A_l.

When the diagnoser is in a positive state, it is certain that a fault has occurred. On the other hand, if the diagnoser is in a negative state, the fault does not occurred. However, if the diagnoser is in a uncertain state, it is not sure if the fault event occurred or not. Then, the existance of a cycle formed only with uncertain states, where the diagnoser can remain forever, will be impossible for it to diagnose the fault occurrence. On the other hand, if it is possible for the diagnoser leaves this uncertain states cycle, then this cycle is not indeterminate. Considering this distinct situations, in order to verify the system diagnosability using a diagnoser, it is necessary to search for indeterminate cycles in G_d [6]. In the sequel we present the definitions of path, cycles and indeterminate cycles.

Definition 2.15 *A path in G is a sequence* $(x_1, \sigma_1, x_2, \ldots, \sigma_{n-1}, x_n)$, *where* $x_i \in X$, $\sigma_i \in \Sigma$, *and* $x_{i+1} = f(x_i, \sigma_i)$, $i = 1, 2, \ldots, n-1$, *and the path is said to be cyclic if* $x_1 = x_n$.

Definition 2.16 *(Cycle) A cycle of G is the set formed of the states of a cyclic path* $(x_k, \sigma_1, x_{k+1}, \sigma_2, ..., \sigma_l, x_{k+l})$, *where* $x_{k+l} = x_k$.

Definition 2.17 *(Indeterminate Cycle) An indeterminate cycle of* G_d *is a set* $\{x_{d_1}, x_{d_2}, ..., x_{d_p}\} \subseteq X_d$ *formed of uncertain states, satisfying the following conditions:*

(i) $\{x_{d_1}, x_{d_2}, ..., x_{d_p}\}$ *forms a cycle in* G_d.

(ii) $\exists(x_l^{k_l}, Y), (\tilde{x}_l^{r_l}, N) \in x_{d_l}$, *with* $x_l^{k_l}$ *not necessarily distinct from* $\tilde{x}_l^{r_l}$, $l = 1, 2, ..., p$, $k_l = 1, 2, ..., m_l$, *and* $r_l = 1, 2, ..., \tilde{m}_l$, *such that the states* $\{x_l^{k_l}\}$, $l = 1, 2, ..., p$, $k_l = 1, 2, ..., m_l$, *and* $\{\tilde{x}_l^{r_l}\}$, $l = 1, 2, ..., p$, $k_l = 1, 2, ..., m_l$, *can be rearranged to form cycles in* G.

Example 2.9 *Consider the system modeled by automaton* G*, shown in Figure 2.9, where* $\Sigma = \{a, b, c, \sigma_u, \sigma_f\}$, $\Sigma_o = \{a, b, c\}$ *and* $\Sigma_{uo} = \{\sigma_u, \sigma_f\}$. *Automaton* $G_l = G \| A_l$ *and diagnoser automaton* $G_d = Obs(G_l)$ *are depicted, respectively, in Figures 2.10 and 2.11. Note that state* $\{6N, 5Y\}$ *is an uncertain state, and* $f_d(\{6N, 5Y\}, b) = \{6N, 5Y\}$*, then uncertain state* $\{6N, 5Y\}$ *forms a cycle in* G_d*. Moreover, associated with components* $\{6N\}$ *and* $\{5Y\}$ *of* $\{6N, 5Y\}$*, exist in* G*, respectively, two cycles formed by states* 6 *and* 5*. Thus,* $\{6N, 5Y\}$ *forms an indeterminate cycle in* G_d.

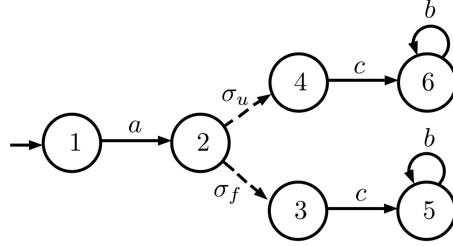


Figure 2.9: Automaton G of Example 2.9.

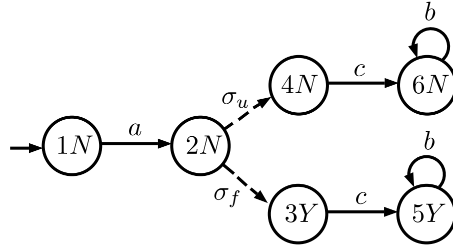


Figure 2.10: Automaton $G_l = G||A_l$ of Example 2.9.

Figure 2.11: Diagnoser automaton $G_d = Obs(G||A_l)$ of Example 2.9.

According to Definition 2.17, a necessary and sufficient condition for language diagnosability, based on the construction of diagnoser automaton G_d, can be stated as follows.

Theorem 2.1 *[6] Language L, generated by automaton G, is diagnosable with respect to projection P_o and $\Sigma_f = \{\sigma_f\}$ if, and only if, G_d does not have indeterminate cycles.*

The following example illustrates the construction of G_d and the verification of the diagnosability of language L.

Example 2.10 *Consider the system modeled by automaton G, shown in Figure 2.12, where $\Sigma = \{a, b, d, g, t, \sigma_f\}$, $\Sigma_o = \{a, b, d, g, t\}$ and $\Sigma_{uo} = \Sigma_f = \{\sigma_f\}$. Suppose that we want to verify if the language of the system L is diagnosable with respect to P_o and σ_f. In order to do so, we need to compute the diagnoser automaton G_d. First, we compute automaton $G_l = G \| A_l$, whose diagram is depicted in Figure 2.13. Then, diagnoser automaton $G_d = Obs(G_l)$, depicted in Figure 2.14, can be computed. Examining G_d, it can be noted that its state set is formed with two negative states ($\{11N\}$ and $\{12N\}$); five uncertain states ($\{1N, 2Y\}, \{7N, 8Y, 3Y\}, \{11N, 9Y, 4Y\}, \{12N, 10Y, 5Y\}$ and $\{11N, 3Y\}$); and three positive states ($\{4Y\}, \{5Y\}$ and $\{6Y\}$). Since G_d does not have indeterminate cycles, L is diagnosable with respect to P_o and $\Sigma_f = \{\sigma_f\}$.*

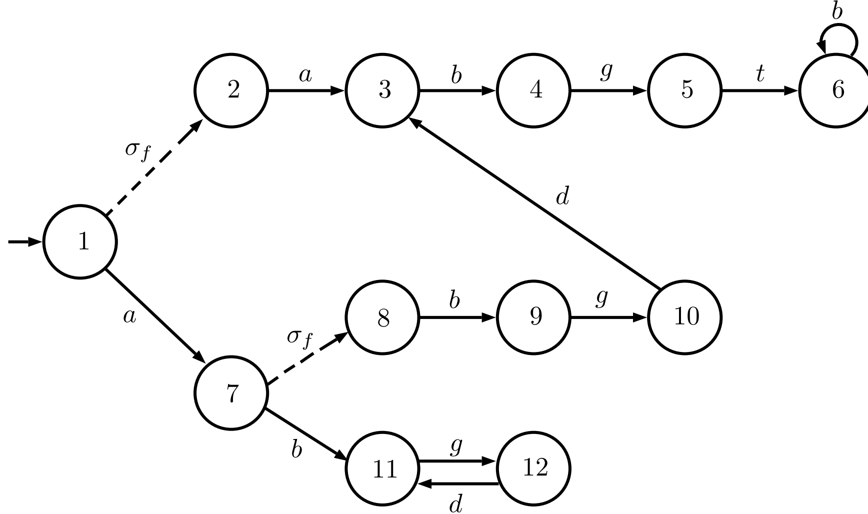


Figure 2.12: Automaton G of Example 2.10.

If L is diagnosable with respect to P_o and Σ_f, then it is possible to define the *delay bound for diagnosis*, denoted as z^*, as the minimum value of z that satisfies the diagnosability condition of Definition 2.14. The following example illustrates the concept of delay bound for diagnosis.

Example 2.11 *Consider diagnoser G_d from Example 2.10. While the system reports to the diagnoser the sequence abgd, the diagnoser is still uncertain if the fault*

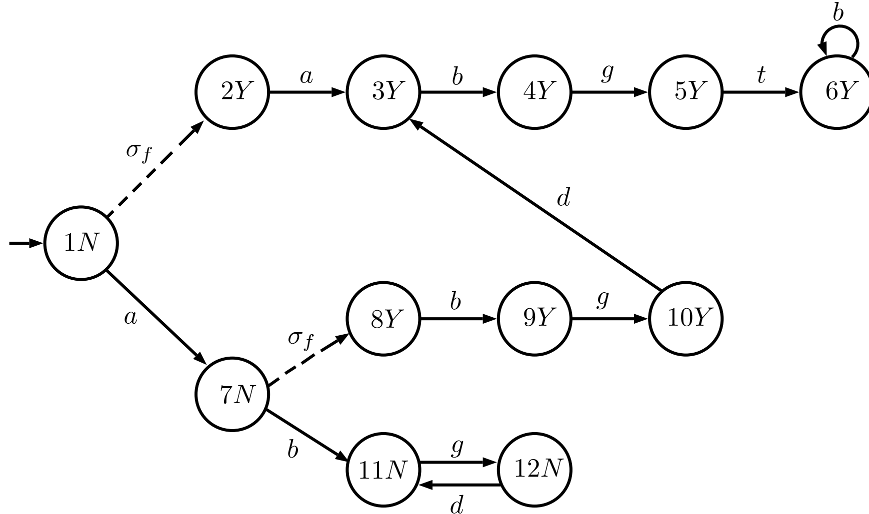


Figure 2.13: Automaton $G_l = G||A_l$ of Example 2.10.

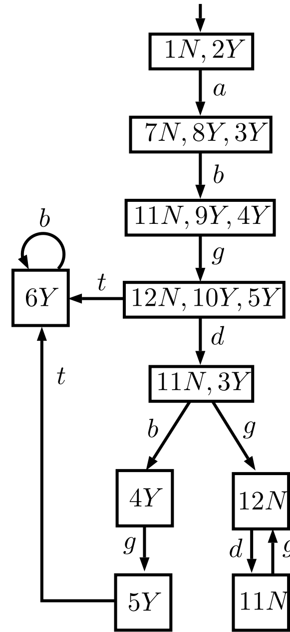


Figure 2.14: Diagnoser automaton $G_d = Obs(G||A_l)$ of Example 2.10.

occurred or not. If sequence abgt occurs, the system is in state 6 *and diagnoser is certain that a fault occurred, which means* $z_1 = 4$*. If sequence abgdb occurs, the system is in state* 4 *and diagnoser is certain that a fault occurred, which means* $z_2 = 5$*. Finally, if sequence abgdg occurs, then the system is in state* 5 *and diagnoser is certain that a fault occurred, which means* $z_3 = 5$*. Then, the minimum value of* z *that satisfies the diagnosability condition of Definition 2.14 is* $z_1 = 4$*.*

In next chapter we present a brief resume of supervisory control theory, and the methods for supervisor reduction.

Chapter 3

Supervisor Reduction for DES

The idea of supervisory control in the modeling of a DES first appeared in [14], being expressed in terms of observation and disabling of controllable events, carried out by a minimally restrictive supervisor on a plant. This supervisor is, roughly speaking, a feedback control capable of changing the behavior of a DES. The uncontrolled behavior of this DES can be modeled by an automaton, whose language usually contains sequences of events that violate certain specifications (or conditions) that are imposed on the system. When this occurs, it is necessary to restrict the behavior of the DES to a subset of this language through the use of feedback control. Then, the supervisor can enable or disable events (usually not all events) in such a way that this restriction is possible.

Modifying the behavior of system modeled by automaton G means, in practice, to restrict language $L(G)$ to some subset of it (namely the *specification*), represented by an automaton H. In order to do so, we must build S (*supervisor*), capable of making this change. However, attempts to apply the supervisory control theory (SCT) to industrial problems have encountered some barriers as state explosion [15]. One way of dealing with these problems is to reduce the size of the supervisor, maintaining the controllability property [1, 2]. In this chapter we will review the basic concepts of SCT and present algorithms to supervisory reduction proposed in the literature, which can be adapted to handle the diagnoser reduction problem.

Although both diagnosers and supervisors are automata that, when reduced, some property of interest must be maintained, there is a fundamental difference considering each case: diagnosers do not have marked states; therefore, the problem concerning the marked language of the automaton is applied only to supervisors, not diagnosers. This suggests a reduced diagnoser is simpler to obtain.

This chapter is organized as follows: in Section 3.1, we present the definition for the behavior of a controlled DES under full observation, where both controllable and uncontrollable events are observable; also, the definition of controllability of a language is presented. In Section 3.2, we explain the algorithms for supervisor

reduction proposed in [2] and [1], with some examples of their applications.

3.1 Controlled DES Under Full Observation

Let L be the language that models the behavior of a given DES, defined over the same event set Σ, where $L = \overline{L}$ is the set of all event sequences the DES can generate. Without loss of generality, let us suppose that $L(G) = L$, and consider that $G = (X, \Sigma, f, x_0, X_m)$ is the automaton that models this DES. Similarly consider the specification automaton H, whose language $L(H) = L_a \subseteq L$ contains only the desired sequences of G, excluding the sequences whose occurrence we want to restrict. Therefore, we can say that L_a is a sublanguage of L.

Then, we can formulate the supervisory control problem as follows: given an automaton G, obtain the supervisor S that interacts with G under a feedback control system, depicted in Figure 3.1. In the design of feedback system, we ensure that automaton S/G (S controlling G) will generate language $L(S/G) = L_a \subseteq L$. Automaton S/G is actually the specification automaton H.

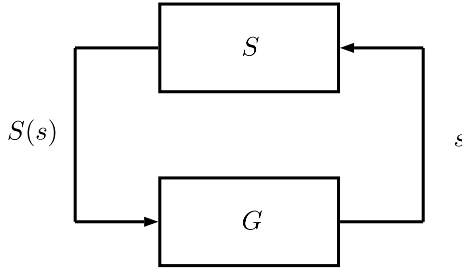


Figure 3.1: Feedback supervisory control structure where S/G.

Formally, a supervisor S is a function of the language generated L by G, whose image is the power set of events Σ, and can be defined as:

$$\begin{aligned} S : L &\rightarrow 2^{\Sigma} \\ s &\rightarrow S(s), \end{aligned}$$

where the active event set $\Gamma_N[f(x_0, s)]$ that G can execute in the state $f(x_0, s)$ is given by:

$$\Gamma_N[f(x_0, s)] = \Gamma[f(x_0, s)] \cap S(s).$$

Thus, G is not capable of execute the event in the state $f(x_0, s)$ if this event does not belong to $S(s)$. In other words, the supervisor S is the control law, while the set $S(s)$ is the control action generated by this law for a given sequence s.

Therefore, the language generated and marked by the controlled system (denoted by S/G) can be defined as follows.

Definition 3.1 *(Language generated and marked by S/G) The language generated by S/G can be recursively defined as*

(i) $\varepsilon \in L(S/G)$*;*

(ii) $s\sigma \in L(S/G) \Leftrightarrow (s \in L(S/G)) \wedge (s\sigma \in L(G)) \wedge (\sigma \in S(s))$.

The language marked by S/G is defined as

$$L_m(S/G) := L(S/G) \cap L_m(G).$$

According to Definition 3.1, it can be noted that the language $L(S/G)$ is prefix-closed.

The solution of the supervisory control problem when there are not uncontrollable and unobservable events always exists and it is relatively simple to obtain. However, in practical systems, there is no guarantee that all events of a system are in fact controllable. Therefore, the event set Σ of the automaton G has to be partitioned in two disjoint subsets as follows:

$$\Sigma = \Sigma_c \dot{\cup} \Sigma_{uc},$$

where

i) Σ_c is the controllable events set, *i.e.*, the events that can be disabled by S;

ii) Σ_{uc} is the uncontrollable event set, *i.e.*, the events that can not be disabled by S.

The representation of a controllable event in a given state transition diagram of an automaton is made with a single trace on the correspondent oriented arc. Unless indicated otherwise in the automaton, the event will be considered uncontrollable.

Then, considering this partition on Σ, a supervisor S will be called *admissible* if

$$\left(\forall s \in L(G), \Sigma_{uc} \cap \Gamma[f(x_0, s)] \subseteq S(s)\right).$$

The supervisor S will be admissible if cannot disable uncontrollable active events. In this work, only admissibe supervisors will be considered.

To assure the synthesis of an admissible supervisor S, we have to consider the possibility that some uncontrollable events must be disabled during the processo in obtaining the specification H. Then, a necessary and sufficient condition has to be presented that guarantee the existance of the supervisor S. This condition derives from the concept of controllability, that will be defined in the following subsection.

3.1.1 Controllability of a Language

Consider a DES modeled by an automaton G controlled by a supervisor S, where all events executed by G are observable by S. Then, the formal definition of controllability is presented as follows.

Definition 3.2 *(Controllability) Let K and L the languages defined over an event set $\Sigma = \Sigma_c \dot{\cup} \Sigma_{uc}$, where $K \subseteq L$ and $L = \overline{L}$. Then, K will be controllable with respect to L and Σ_{uc} if, and only if*

$$\overline{K}\Sigma_{uc} \cap L \subseteq \overline{K}.$$

For all sequences $s \in \overline{K}$ and an uncontrollable event $\sigma_{uc} \in \Sigma_{uc}$, if $s\sigma_{uc} \in L$, then also $s\sigma_{uc} \in \overline{K}$. According to Definition 3.2, controllability is a prefix-closed property of a language, *i.e.*, K will be controllable if, and only if, $\overline{K}$ is also controllable. Moreover, although controllability is not preserved under intersection, it is preserved under union. In other words, if two languages K_1 and K_2 are controllable, then $K_1 \cup K_2$ is also controllable.

The existence of a supervisor whose language is controllable is conditioned to the concept of controllability. Considering this, it is necessary to verify the controllability of a language, which can be stated by the following theorem.

Theorem 3.1 *(Controllabiliy theorem [13]) Consider a DES described by automaton $G = (X, \Sigma, f, x_0, X_m)$, where $\Sigma_{uc} \subseteq \Sigma$ and $L(G) = L = \overline{L}$. Let K be the specification of the language such that $K \subseteq L$, being $K \neq \emptyset$. Then, exists a supervisor S, such that $L(S/G) = \overline{K}$ if, and only if*

$$\overline{K}\Sigma_{uc} \cap L \subseteq \overline{K}.$$

Thus, for K to be controllable with respect to L and Σ_{uc} all undesirable sequences must be part of the specification.

In order to illustrate the concept of controllability, consider the following example.

Example 3.1 *Let G and H be two automata depicted in Figures 3.2 and 3.3, respectively, where $\Sigma = \{a, b, c\}$, $L = L(G) = \overline{\{abc, acb\}}$ and $K = L(H) = \{acb\}$. Also, let $\Sigma_{uc} = \{b\}$. In state 2, the supervisor must disable event b so only event c could occur, in order to satisfy the specification. However, event b is uncontrollable, which makes this impossible to occur. Therefore, it is said that K (and also $\overline{K}$) is not controllable with respect to L and Σ_{uc}. On the other hand, if $\Sigma_{uc} = \{c\}$, the system will be controllable.*

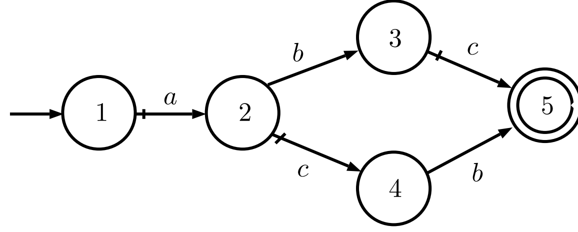


Figure 3.2: Automaton G of Example 3.1.

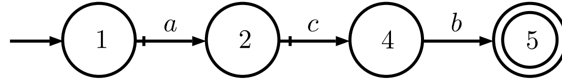


Figure 3.3: Automaton H of Example 3.1.

Supremal Controllable Sublanguage

When the specification language K is uncontrollable, one of the alternatives is to find the "largest" sublanguage of K that is controllable, where "largest" concerns inclusion of sets. This language is called *supremal controllable sublanguage* of K, denoted by $K^{\uparrow C}$.

According to [16], the controllability property is closed under set union, *i.e.*, there exists a unique largest controllable language $K^{\uparrow C}$ such that $K^{\uparrow C} \subseteq K$. In other words, if languages $K_i, i = 1, 2, \ldots, n$, are controllable with respect to $L(G)$ and Σ_{uc}, then the language $K_1 \cup K_2 \cup \ldots \cup K_n$ is also controllable. Then, it can be concluded that the supremal controllable sublanguage of a given language K always exists. The following properties of this supremal controllable sublanguage hold [13]: (i) In the worst case, $K^{\uparrow C} = \emptyset$; ($ii$) If K is controllable, then $K^{\uparrow C} = K$; (iii) If K is prefix-closed, then so is $K^{\uparrow C}$.

The following example illustrates the concept of $K^{\uparrow C}$.

Example 3.2 *Consider the two automata, G and H, depicted in Figures 3.4 and 3.5, respectively, where $\Sigma = \{a, b, c\}$ and $\Sigma_{uc} = \{a\}$. Let $L = L(G)$ and $K = L_m(H)$. Then,*

$$K = \{abc, acb, cba\}.$$

Note that K is not controllable, considering that contains as a prefix event c, which can be extended in L by the uncontrollable event a, and $ca \notin \overline{K}$. Thus, we need to remove from K all sequences that contain event c as a prefix. After removing those sequences from K, we obtain

$$K_1 = \{abc, acb\},$$

which is controllable. Then, $K^{\uparrow C} = K_1$.

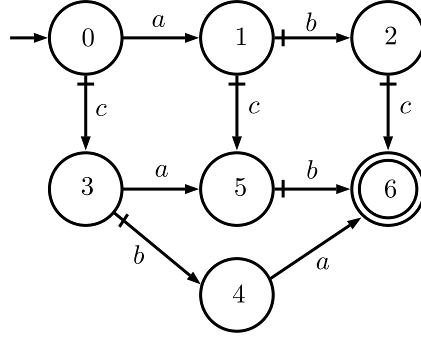


Figure 3.4: Automaton G of Example 3.2.

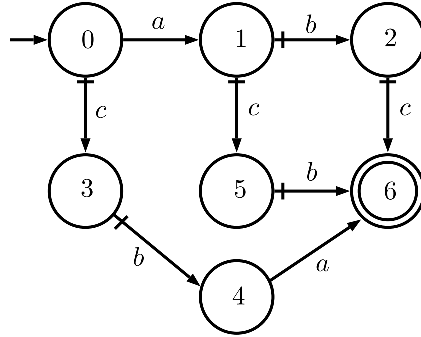


Figure 3.5: Automaton H of Example 3.2.

In order to apply the supervisor reduction algorithms proposed in [1] and [2] in diagnosers, the marked states of supervisors will be omitted. In next section, we present these algorithms.

3.2 Supervisor Reduction Methods

Considering that supervisors normally contain some redundant information that can be easily inferred from the structure of the plant, it seemed natural to conceive the possibility that the construction of a minimal (or at least with fewer states than the original) could be made. In [11], the authors stated that the construction of a minimal supervisor is time-exponential with respect to the state size of a given supervisor. It was shown in [2] that this problem is, in fact, NP-hard. Then, in [1] is proposed a heuristic polynomial-time algorithm for supervisor reduction, based on the concept of control covers. This method is presented in the next section.

3.2.1 Reduction Based on Control Covers

The supervisor reduction problem in [1] can be resumed as follows: given a supervisor $S = (X^S, \Sigma, f^S, x_0^S, X_m^S)$ for a specification H on a plant $G = (X, \Sigma, f, x_0, X_m)$, we

want to find a supervisor S' with fewer states than S and that is control equivalent to S, *i.e.*, the language of the supervisory control of S is equivalent to the language of the supervisory control of S'. Therefore, such an automaton S' is called a reduced supervisor.

In order to compute the control cover, the following definitions were proposed in [1].

Definition 3.3 *(Eligible event set) The set of eligible events at any state $x \in X$ is defined as*

$$Elig(G, x) := \{\sigma \in \Sigma | f(x, \sigma)!\}.$$

Definition 3.4 *(Cover) A cover of a supervisor S is defined to be a family $\{X_i \subseteq X^S : i \in I\}$ of the subsets of the state set of S with the following properties:*

1. $(\forall i \in I), X_i \neq \emptyset$
2. $(\forall i, j \in I), i \neq j \Rightarrow X_i \nsubseteq X_j$
3. *for a subset $I_m \subseteq I$: $X_m^S = \cup_{i \in I_m} X_i \rightarrow X^S - X_m^S = \cup_{i \in I - I_m} X_i$, where I is some arbitrary index set.*

According to Definition 3.4, a cover of a set X^S is a family of subsets of X^S whose union is X^S. Also, the elements of a cover of S inherit the marking structure of S.

Definition 3.5 *(Deterministic cover) A cover of S is defined to be deterministic if*

$$\left[(\forall i, j \in I)\,(\forall \sigma \in \Sigma)\,(\exists x, y \in X_i)\,|f^S(x, \sigma) \in X_j \text{ and } f^S(y, \sigma)! \Rightarrow f^S(y, \sigma) \in X_j\right].$$

where f^S is the transition function of S.

According to Definition 3.5, a cover is deterministic if no two states of any element of the cover make transitions to different elements of the cover under the same event.

Definition 3.6 *(Disabled set) Let $x_1 \in X^S$ be any state of S. Let $X'(x_1) \subseteq X$ be a subset of states of the plant defined as follows:*

$$X'(x_1) = \{x \in X : (x, x_1) \in X^{G \| S}\}.$$

Then, the set of disabled events at x_1 is defined as

$$Disabled(x_1) := \{\sigma \in \Sigma_c : (\exists \in X'(x_1))[\sigma \in Elig(G, x) - Elig(S, x_1)]\}.$$

The set $Disabled(x_1)$ is formed of the events that are disabled in the plant at a state $x \in X$ such that $(x, x_1) \in X^{G\|S}$.

Definition 3.7 *(Control cover) Any two states $x_1, x_2 \in X^S$ are defined to be control consistent if $Disabled(x_1) = Disabled(x_2)$. A cover of S is defined to be control consistent if*

$$(\forall i \in I)(\forall x_1, x_2 \in X_i)[x_1 \text{ and } x_2 \text{ are control consistent}].$$

If a cover C of S is deterministic and control consistent, then C is called a control cover.

Before we present the application of the method proposed in [1], some definitions are necessary.

Definition 3.8 *(Ineligible event set) Let $x \in X^S$. Then, the set of ineligible events at x is defined as*

$$Ineligible(x) := \Sigma_c - (Elig(S, x) \cup Disabled(x)).$$

The set $Ineligible(x)$ is formed of the events that are not physically possible at x.

The difference between the disabled set and ineligible set is that the later is the set of events that are physically impossible to occur in a given state of plant G due to the nature of the system.

Example 3.3 *Let G be an automaton that represents a plant to be controlled, depicted in Figure 3.6, and S be a supervisor controlling G, depicted in Figure 3.7. Consider that $\Sigma = \{a, b, c, d, g, t\}$, where $\Sigma_c = \{a, b, d, g, t\}$ and $\Sigma_{uc} = \{c\}$. According to Definition 3.3, the eligible event set for state 1 is $\{g, t\}$. Note that state 1 in S does not disable any event in G, and the events a, b, d are ineligible to occur in this state, according to Definition 3.8. Also note that states 4 and 5 in S disable event d in G. Then, the ineligible events set of states 4 and 5 are, respectively, $\{a, b, g, t\}$ and $\{b, g, t\}$. Table 3.1 shows the disabled and the ineligible state sets for all states of S.*

According to Definition 3.5, a possible deterministic cover for S is the set $C = \{\{1, 4, 5\}, \{2, 3, 6\}\}$, where does not exist two states in any of the subsets $\{1, 4, 5\}$ and $\{2, 3, 6\}$ that make transitions to different elements of the cover under the same event.

Definition 3.9 *(Control compatible) Any two states $x_1, x_2 \in X^S$ are defined to be control compatible if*

$$Disabled(x_1) \subseteq Disabled(x_2) \cup Ineligible(x_2), \text{ and}$$

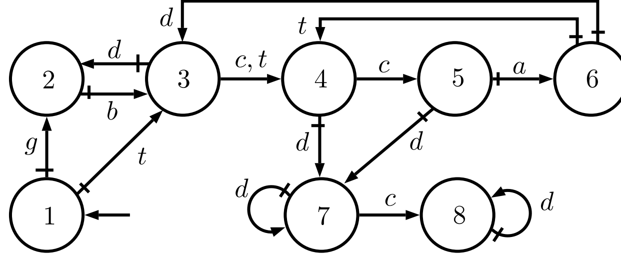


Figure 3.6: Plant G of Example 3.4.

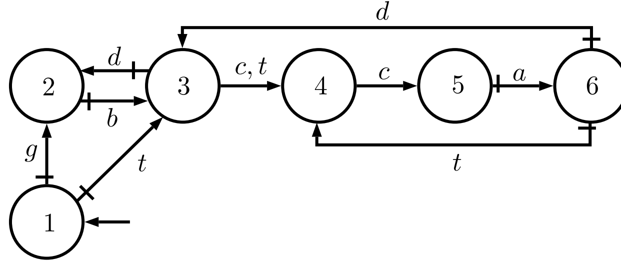


Figure 3.7: Supervisor S of Example 3.4.

Table 3.1: Disabled and ineligible state sets for all states of S of Example 3.4.

State	Disabled Events	Ineligible Events
1	$\emptyset$	$\{a, b, d\}$
2	$\emptyset$	$\{a, d, g, t\}$
3	$\emptyset$	$\{a, b, g\}$
4	$\{d\}$	$\{a, b, g, t\}$
5	$\{d\}$	$\{b, g, t\}$
6	$\emptyset$	$\{a, b, g\}$

$$Disabled(x_2) \subseteq Disabled(x_1) \cup Ineligible(x_1).$$

Definition 3.10 *(Marking compatible) Any two states $x_1, x_2 \in X^S$ are defined to be marking compatible if*

$$(\forall x \in X_m^S)[((x, x_1), (x_{,2}) \in X^{G\|S}) \Rightarrow (x_1 \in X_m^S \leftrightarrow x_2 \in X_m^S)].$$

Definition 3.11 *(Compatible states) Any two states $x_1, x_2 \in X^S$ are defined to be compatible if they are control and marking compatible. Otherwise, the two states are defined to be incompatible. For any $x \in X^S$, the sets of compatible and incompatible states are defined as*

$$Compatible(x) := \{y \in X^S : x \text{ and } y \text{ are compatible}\}$$

$$Incompatible(x) := \{y \in X^S : x \text{ and } y \text{ are not compatible}\}.$$

Definition 3.12 *(Mergeable states) Any two states* $x_1, x_2 \in X^S$ *are defined to be mergeable if they are compatible and*

$$(\forall \sigma \in \Sigma)[f^S(x_1, \sigma) \text{ and } f^S(x_2, \sigma) \text{ are compatible}].$$

For any $x \in X^S$*, the set of mergeable states is defined as*

$$Mergeable(x) := \{y \in X^S : x \text{ and } y \text{ are mergeable}\}.$$

According to Definitions 3.11 and 3.12, it can be noted that two states will be considered mergeable if they are compatible, and their transitions functions under the action of the same events lead to also compatible states. Thus, the cover computed where its elements are mutually mergeable states will be deterministic if the original transition structure of the supervisor is respected.

In [1], three algorithms were developed in order to compute the control cover and merge states in the supervisor. The first one is to compute the mergeable state sets for each pair of events, according to Definition 3.12, and is called *findMergeableStateSets*. Thus, it is necessary to find all the compatible and incompatible pairs in order to compute the mergeable set. The algorithm starts creating a list for each pair (x, y) that is reached for another pair through the execution of the same event. Then, the next step is to cross all the pairs (x, y) such that $y \in Incompatible(x)$. The "crossing" operation means that the pair crossed is not mergeable and, therefore, will not compose the desired control cover. Next, the algorithm checks if there are any pair of states that reaches a crossed pair, and then cross this pair. In other words, for all $\sigma \in \Sigma$ and $(w, z) \in X^S$ such that $f^S(w, \sigma) = x$ and $f^S(z, \sigma) = y$ or vice-versa: if (x, y) is crossed, then the pair (w, z) will also be crossed. This occurs recursively until all pairs on the list of each pair crossed are also crossed. The other pairs that were not crossed in this procedure are added to $Mergeable(x)$, which is the set of states that can be mergeable with state x. The algorithm ends when the mergeable set is obtained for all states in X^S.

The second algorithm, called *findMaximalMutuallyMergeableSet*, begins by computing an element $X^S_{i_0}$ that contains the initial state of S. Consider that α is an event eligible to occur at any state belonging to $X^S_{i_0}$; then, to guarantee that the computed cover is deterministic, it is necessary to ensure that $f^S(X^S_{i_0}, \alpha)$ is a subset of some element of the cover. Otherwise, the algorithm computes an element that contains it. The main procedure of this algorithm takes as input a set whose elements are pairwise mergeable, and the output X^S_i is also a set such that all its elements are pairwise mergeable. This maximal element, X^S_i, will form an element of the control cover if it is not a subset of any existing element. Therefore, the criterion of merging states in [1] can be stated: the maximal mutually mergeable

state set, where all its elements are pairwise mergeable, is put on the control cover C and is automatically merged in order to form the reduced supervisor S'. It can be noted that this procedure always starts computing the set X_{i_0}, where the initial state of S is the first element of this set. Thus, if this set is the maximal mutually mergeable set, it can be inferred that the reduced supervisor depends of the choice of the initial state of S.

The last and third algorithm, called *findControlCover*, puts exactly this maximal element X_i^S as the first state set (not necessarily containing the initial state of S) of the cover. Then, it computes the maximal element considering the other states that were not added to X_i^S, and adds the result to C. The algorithm ends when all the states forms maximal elements in the resulting control cover, and the reduced supervisor is obtained by the merging of the elements that forms each subset of the control cover.

The following example illustrates the supervisor reduction method proposed in [1].

Example 3.4 *Let us consider again automaton G from Example 3.3 that represents a plant to be controlled, depicted in Figure 3.6, and S be a supervisor controlling G that we want to reduce, depicted in Figure 3.7. Consider again that $\Sigma = \{a, b, c, d, g, t\}$, where $\Sigma_c = \{a, b, d, g, t\}$ and $\Sigma_{uc} = \{c\}$. Then, the Table 3.1 that shows the disabled and the ineligible state sets for all states of S obtained in Example 3.3 can be computed.*

Note that event d is eligible to occur in state 3, but it is disabled in state 4. Then, states 3 and 4 are incompatible. Also note that event d is disabled in state 5. Thus, states 3 and 5 are incompatible. From Table 3.1, we can compute the compatible state sets for each state, shown in Table 3.2.

Table 3.2: Compatible states of Example 3.4.

State	Compatible States
1	$\{1, 2, 3, 4, 5, 6\}$
2	$\{1, 2, 3, 4, 5, 6\}$
3	$\{1, 2, 3, 6\}$
4	$\{1, 2, 4, 5\}$
5	$\{1, 2, 4, 5\}$
6	$\{1, 2, 3, 6\}$

According to Table 3.2 of compatible state sets, we now compute the first algorithm, findMergeableStateSets. *The algorithm starts with initial state 1, and does the following checks and crossing:*

- $(f^S(1, t), f^S(3, t)) = (3, 4)$. *Since states 3 and 4 are not compatible, then $(1, 3)$ is crossed;*

- $(f^S(1,t), f^S(6,t)) = (3,4)$. *Since states* 3 *and* 4 *are not compatible, then* $(1,6)$ *is crossed;*

Since there are no more pair of compatible states that execute the same event, the algorithm ends and the mergeable states are computed. The result is shown in Table 3.3.

Table 3.3: Mergeable states of Example 3.4.

State	Mergeable States
1	1,2,4,5
2	1,2,3,4,5,6
3	2,3,6
4	1,2,4,5
5	1,2,4,5
6	2,3,6

Now, the algorithm findControlCover starts, where the first step is to compute the algorithm findMaximalMutuallyMergeableSet. This algorithm starts with initial state 1*:* $X_{i_0}^S := findMaximalMutuallyMergeableSet(\{1\})$. *Since all elements of* $Mergeable(1)$ *are pairwise mergeable, the output is* $X_{i_0}^S = Mergeable(1) = \{1,2,4,5\}$. *The events eligible at* $X_{i_0}^S$ *are:* $\{a,b,c,g,t\}$. *Then, the algorithm tests the transitions for all eligible events at* $X_{i_0}^S$*, obtaining the following:*

- $f^S(X_{i_0}^S, a) = \{6\}$;
- $f^S(X_{i_0}^S, b) = \{3\}$;
- $f^S(X_{i_0}^S, c) = \{5\}$;
- $f^S(X_{i_0}^S, g) = \{2\}$;
- $f^S(X_{i_0}^S, t) = \{3\}$;

When $f^S(X_{i_0}^S, \sigma)$ *is a state (or set of states) that are in* $X_{i_0}^S$*, the algorithm continues without any new merges. This is the case for* $f^S(X_{i_0}^S, c) = \{5\}$ *and* $f^S(X_{i_0}^S, g) = \{2\}$. *Then it is tested the output for* $findMaximalMutuallyMergeableSet(\{3\})$ *whose result is:*

$$findMaximalMutuallyMergeableSet(\{3\}) = \{2,3,6\}$$

Note that for state, 2*,* 3 *and* 6*, we have* $f^S(X_{i_1}^S, d) = \{2,3\}$*, which is an element of set* $\{2,3,6\}$. *Then,* $\{2,3,6\}$ *is added to* C. *Since there are no more states reached with the same* σ *from* $\{2,3,6\}$*, the iteration for* $X_{i_1}^S$ *ends. After this iteration, the algorithm discovers that no new states are generated.*

Then, the algorithm findControlCover simply puts the sets formed in the previous algorithm on C:

$$C = \{\{1,2,4,5\},\{2,3,6\}\}\,.$$

Thus, the supervisor induced by C, S', is depicted in Figure 3.8.

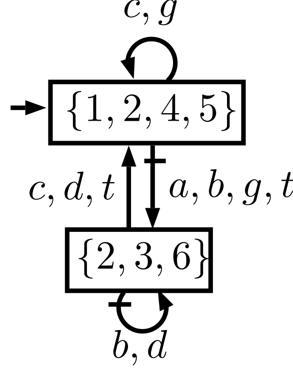


Figure 3.8: Reduced supervisor S' of Example 3.4.

Note that the reduced supervisor S' of Figure 3.8 is nondeterministic. Therefore, in the following section, the method for supervisor reduction proposed in [2] is presented, where the concept of control congruence is introduced in order to guarantee that the reduced supervisor is deterministic.

3.2.2 Reduction Based on Control Congruence

As addressed in [2], the optimal solution for the supervisory control problem is obtained by the supremal controllable sublanguage, represented by the supremal supervisor, which has state size of order the product of state sizes of the plant and specification transition structures. Therefore, the state size reduction of the supremal supervisor can be possible without affecting controlled behavior. The definition of a cover in [2] is less restrictive than that of [11], showing that a cover is necessary as well as sufficient for supervisor reduction. Thus, the authors in [2] developed the concept of control congruence, which is a special case of a control cover, providing a polynomial-time reduction algorithm.

In this section, consider that the DES to be controlled is modelled as a discrete transition structure $G = (X, \Sigma, f, x_0, X_m)$ and $S = (X^S, \Sigma, f^S, x_0^S, X_m^S)$ is a supremal supervisor. Let $E : X^S \to 2^\Sigma$ denote the enabled event set at state $x \in X^S$, with

$$x \mapsto E(x) := \{\sigma \in \Sigma | f^S(x,\sigma)!\}.$$

Also, let $D : X^S \to 2^\Sigma$ denote the disabled event set at state $x \in X^S$, with

$$x \mapsto D(x) := \{\sigma \in \Sigma | f^S(x, \sigma)! \text{ and } (\exists s \in \Sigma^*)[f^S(x_0^S, s) = x \text{ and } f(x_0, s\sigma)!]\}.$$

Concerning the marked behavior of S, let $M : X^S \to \{true, false\}$ and $T : X^S \to \{true, false\}$, with

$$x \mapsto M(x) := \{true, \text{ if } x \in X_m^S\},$$

$$x \mapsto T(x) := \{true, \text{ if } (\exists s \in \Sigma^*) f^S(x_0^S, s) = x \text{ and } f(x_0, s) \in X_m\}.$$

Let also $\mathcal{R} \subseteq X^S \times X^S$ be the binary relation such that for a pair $x, x' \in X^S$, $(x, x') \in \mathcal{R}$ if, and only if:

C1. $E(x) \cap D(x') = E(x') \cap D(x) = \emptyset$.

C2. $T(x) = T(x') \Rightarrow M(x) = M(x')$.

Condition **C1** says that for a pair of states $(x, x') \in \mathcal{R}$, the associated enable/disable control actions should be consistent, *i.e.*, no event is enabled at x but disabled at x'. Condition **C2** requires that states $(x, x') \in \mathcal{R}$ be consistently marked either *true* or *false* in S if they are reachable by some sequences s, s' in $L_m(G)$, or else if neither is reachable by sequences in $L_m(G)$.

The definition of control congruence is presented as follows.

Definition 3.13 *(Control congruence [2]) A cover $C = \{X_i^S \subseteq X^S | i \in I\}$ of X^S, where I is an index set, is a control cover on S if*

1. $(\forall i \in I) X_i^S \neq \emptyset \wedge (\forall x, x' \in X_i^S)(x, x') \in \mathcal{R}$
2. $(\forall i \in I)(\forall \sigma \in \Sigma)(\exists j \in I)[(\forall x \in X_i^S)\xi(x, \sigma)! \Rightarrow \xi(x, \sigma) \in X_j^S]$

The subsets X_i^S are the cells of C. Then, a control cover C is a control congruence if C is a partition on X^S, namely the X_i^S are pairwise disjoint.

Note that condition **C1** also requires that each cell of C must be nonempty, and that each pair of states in the same cell should belong to $\mathcal{R}$, *i.e.*, both associated control action and marked status should be consistent. This is the same condition stated for the computation of control cover in Section 3.2.1. Then, condition **C2** states that for each $X_i^S \in C$ and each $\sigma \in \Sigma$, the set of states that can be reached from any state in X_i^S by the next transition executing event σ is covered by some $X_j^S \in C$. It can be inferred that in the cells of a control cover a state can belong to more than one cell X_i^S, but this ambiguity is solved by the analysis of the enabled restriction E to the cell X_i^S of the cover, maintaing the pairwise disjointness.

Therefore, the algorithm called *Reduction Algorithm (RA)*, proposed in [2], computes a control congruence as follows. First a control congruence C on S is created, initially set to be a set of one state of X^S. Also, a list on $X^S \times X^S$, starting with the emptyset, is created. This is a list of state pairs that are waiting to be merged, *i.e.*, if all subsequent tests for mergeability return *true*, then each state pair will be merged in the same cell of the control congruence that is formed by RA. Consider that the initial state of S is x_1. The main procedure starts by testing the initial state with the next one by following the order of the states previously given from S. Then, if two states $x_1, x_2 \in X^S$ satisfy $(x_1, x_2) \in \mathcal{R}$, *i.e.*, if the pair (x_1, x_2) satisfies both conditions **C1** and **C2**, the algorithm checks mergeability in the following with a subroutine procedure: for all $\sigma \in \Sigma$, where $f^S(x_1, \sigma)!$ and $f^S(x_2, \sigma)!$, it is verified if $(f^S(x_1, \sigma), f^S(x_2, \sigma)) \in \mathcal{R}$. If $(f^S(x_1, \sigma), f^S(x_2, \sigma)) \notin \mathcal{R}$, then x_1 and x_2 will not be merged on S. On the other hand, if $(f^S(x_1, \sigma), f^S(x_2, \sigma)) \in \mathcal{R}$, then x_1 and x_2 will be merged on S, forming the cell $\{x_1, x_2\}$ of the control congruence C.

If $(x_1, x_2) \notin \mathcal{R}$, then the algorithm returns *false*, and the pair (x_1, x_2) is not merged, *i.e.*, no state-merge occurs on S. This procedure repeats until a pair satisfies both conditions **C1** and **C2**, and the subroutine to check mergeability is computed. After this, the algorithm tests the mergeability between x_1 and next state of S, x_3. Suppose now that x_1 and x_2 were merged on S. Then, it is checked if state x_3 satisfies $x_1, x_3 \in \mathcal{R}$ and $x_2, x_3 \in \mathcal{R}$. If one of them holds not true, then no state is merged on S. On the other hand, if both holds true, the subroutine procedure is computed for all $\sigma \in \Sigma$, where $f^S(x_1, \sigma)!$, $f^S(x_2, \sigma)!$ and $f^S(x_3, \sigma)!$, it is checked if $(f^S(x_1, \sigma), f^S(x_3, \sigma)) \in \mathcal{R}$ and if $(f^S(x_2, \sigma), f^S(x_3, \sigma)) \in \mathcal{R}$. If one of them holds not true, there is no state merging on S. On the other hand, if both holds true, then state x_3 is merged on S with state x_1 and x_2, *i.e.*, x_3 is added to cell $\{x_1, x_2\}$ of C, forming the cell $\{x_1, x_2, x_3\}$.

This procedure ends when all the pairs that are merged are added to control congruence C, and the pairwise disjointness is guaranteed.

The following example illustrates the supervisor reduction method proposed in [2].

Example 3.5 *Let us consider again the plant automaton G and supervisor S controlling G, depicted in Figures 3.6 and 3.7, respectively. Note that S is in fact a supremal supervisor. Consider also that $D(4) = D(5) = \{d\}$, and $D(1) = D(2) = D(3) = D(6) = \emptyset$. Then, the algorithm starts testing the mergeability between initial state 1 and the next state considering the previous order given by S. Thus, the first pair tested for mergeability is $(1, 2)$. It is clear that $(1, 2) \in \mathcal{R}$. Since there are no $\sigma \in \Sigma$ such that $f^S(1, \sigma)!$ and $f^S(2, \sigma)!$, then states 1 and 2 are merged on S, and $(1, 2)$ is added to C.*

Now, the algorithm continues, and checks that $(1,3) \in \mathcal{R}$*. Thus, the algorithm checks that* $f^S(1,t) = 3$ *and* $f^S(3,t) = 4$*, and tests if* $(3,4) \in \mathcal{R}$*. Clearly* $(3,4) \notin \mathcal{R}$*, and then no states are merged on* S*. The next pair tested is* $(1,4)$*, and* $(1,4) \in \mathcal{R}$*. Since there are no events in common between states* 1 *and* 4*, the algorithm now tests state* 4 *with the other state in the cell* $\{1,2\}$ *of the cover. Clearly,* $(2,4) \in \mathcal{R}$*, and then state* 4 *is added to cell* $\{1,2\}$*, generating the cell* $\{1,2,4\}$*. The same procedure is done with state* 5*, where clearly the following holds true:* $(1,5) \in \mathcal{R}$*,* $(2,5) \in \mathcal{R}$ *and* $(4,5) \in \mathcal{R}$*. Thus, the state* 5 *is added to cell* $\{1,2,4\}$*, generating the cell* $\{1,2,4,5\}$*. The last test for state* 1 *is with state* 6*, where* $(1,6) \in \mathcal{R}$*. Thus, the algorithm checks that* $f^S(1,t) = 3$ *and* $f^S(6,t) = 4$*, and tests if* $(3,4) \in \mathcal{R}$*. Clearly* $(3,4) \notin \mathcal{R}$*, and then no states are merged on* S*. Therefore, the last remaining test is for pair* $(3,6)$*. Note that* $(3,6) \in \mathcal{R}$*, but* $f^S(3,d) = 2$ *and* $f^S(6,d) = 3$*. Since* 2 *is already in a cell where state* 3 *cannot be added, the merge* $(3,6)$ *violates the pairwise disjointness of control congruence, and then states* 3 *and* 6 *are not merged. Therefore, the cells* $\{3\}$ *and* $\{6\}$ *are added to* C*, and the control congruence is formed:*

$$C = \{\{1,2,4,5\},\{3\},\{6\}\}.$$

Finally, the induced supervisor, S'*, has its states formed for each cell of the cover* C*. The reduced supervisor* S' *is depicted in Figure 3.9.*

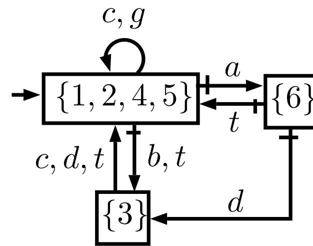


Figure 3.9: Reduced supervisor S' of Example 3.5.

Note that the reduced supervisor for Example 3.4 has two states, but is nondeterministic, while in Example 3.5 the reduced supervisor has three states, and is deterministic. Although the results are satisfactory in terms of complexity reduction (the original S had six states), remains open if it is possible to obtain a smaller deterministic supervisor than the one depicted in Figure 3.9. In next chapter, we present an algorithm for diagnoser reduction and apply it to a diagnoser with similar construction to the supervisor S from the examples of this section, in order to compare the efficiency of methods.

Chapter 4

Diagnoser Reduction Method

An efficient diagnoser is the one with the less delay of diagnosis as possible, so that the fault can be diagnosed without causing considerable impact on the system. There is a drawback of implementing the diagnoser proposed in the literature, which concerns its state set that can be very large, requiring a great amount of memory to be implemented in complex systems. Then, a reduced diagnoser is desirable for greater economy of implementation, as long as the diagnosis delay is the same as the original diagnoser. In this chapter, we propose an algorithm to reduce diagnosers, in order to maintain the diagnosability of language and diagnosis delay.

Consider that language L of an automaton G that models a given DES is diagnosable with respect to P_o and Σ_f. In this case, the diagnoser proposed in [6] can be constructed for online diagnosis. Since the diagnoser is based on the computation of an observer automaton, then, in the worst-case, its state set can grow exponentially with the number of states of the plant G. However, in average, the size of the diagnoser does not grow exponentially with the number of system states, but it can still be a large number for complex and large systems [7].

In this chapter we propose an algorithm for reducing the diagnoser G_d such that the diagnosability of L and the delay for diagnosis are both preserved. The structure of the chapter is as follows. In Section 4.1, the first step in order to reduce a diagnoser is presented. In Section 4.2, we present what is in fact the procedure of state merging. In Section 4.3, we compute, for each state of diagnoser, the set of states that cannot be merged with that state. In Section 4.4, the computation of a deterministic diagnoser is presented, considering our desire to implement deterministic reduced diagnosers. In Section 4.5 we compute the mergeable states set for each pair of states. In Section 4.6 we present the algorithm for diagnoser reduction, with an example of its application.

4.1 First Step in Diagnoser Reduction

The first step for reducing the diagnoser is to merge all positive states of G_d into a single state, denoted as F, generating a new diagnoser automaton $G'_d = (X'_d, \Sigma_o, f'_d, x'_{0_d})$, where X'_d is the set formed of singletons of all elements of $X_N \cup X_{NY} \cup \{F\}$, *i.e.*, $X'_d = \{\{x_d\} : x_d \in X_N \cup X_{NY} \cup \{F\}\}$, $f'_d(\{x_d\}, \sigma) = \{f_d(x_d, \sigma)\}$, if $f_d(x_d, \sigma) \in X_N \cup X_{NY}$, $f'_d(\{x_d\}, \sigma) = \{F\}$, if $f_d(x_d, \sigma) \in X_Y$, and $f'_d(\{F\}, \sigma) = \{F\}$, for all $\sigma \in \Sigma_o$, and $x'_{0_d} = \{x_{0_d}\}$. This state merging is possible since the knowledge of the system state after the fault detection is unnecessary for the diagnosis decision and for the computation of the diagnosis delay. In the sequel, we present an example to illustrate the computation of G'_d.

Example 4.1 *Consider the system modeled by automaton G, shown in Figure 4.1, where $\Sigma = \{a, b, c, d, t, \sigma_f\}$, $\Sigma_o = \{a, b, c, d, t\}$ and $\Sigma_{uo} = \Sigma_f = \{\sigma_f\}$, and suppose that we want to verify if the language of the system L is diagnosable with respect to P_o and σ_f. In order to do so, the diagnoser automaton G_d, depicted in Figure 4.2, can be computed. Since G_d does not have indeterminate cycles, then L is diagnosable with respect to P_o and σ_f.*

The positive states of G_d are $\{7Y\}$ and $\{8Y\}$. Then, we can merge $\{7Y\}$ and $\{8Y\}$ into a single state, denoted as $\{F\}$, generating the diagnoser automaton G'_d, shown in Figure 4.3. Note that a self-loop is introduced in state $\{F\}$ labeled with all observable events.

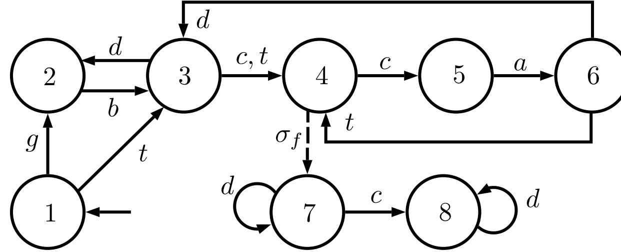


Figure 4.1: Automaton G of Example 4.1.

In the first step of the diagnoser reduction only positive states are merged. However, in some cases, it is possible to reduce G'_d, obtaining a new diagnoser automaton $G^r_d = (X^r_d, \Sigma_o, f^r_d, x^r_{0_d})$, by merging also negative and uncertain states of G'_d. In next section, we present the procedure of state merging.

4.2 Procedure of State Merging

After merging positive states into a single state labeled F, it is possible to reduce even more the original diagnoser. The method proposed in this work consists in

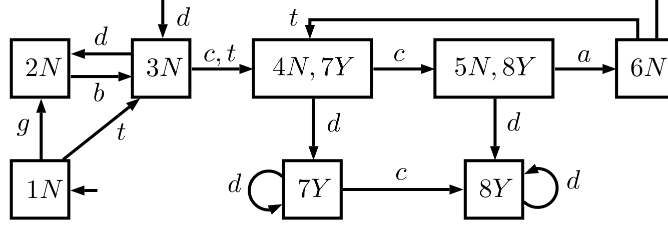


Figure 4.2: Automaton G_d of Example 4.1.

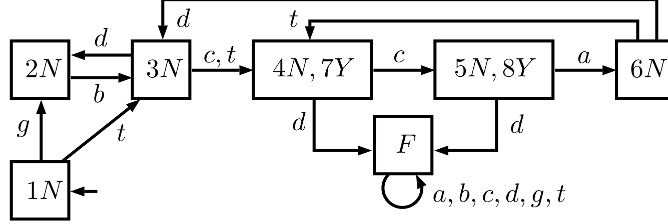


Figure 4.3: Automaton G'_d of Example 4.1.

the merge of negative and uncertain states. Thus, in this section we present an algorithm for merging two states of a given diagnoser.

In Algorithm 4.1, we compute automaton G''_d obtained after merging two states x and y of an automaton $\hat{G}_d$ into a single state $x \cup y$. It is important to remark that all reduced diagnosers computed in this work are obtained from merging states of the original diagnoser G'_d. Thus, since each state of G'_d is a singleton formed of a state of G_d or F, then a state $x''_d \in X''_d$ of G''_d has the following form $x''_d = \{x_{d_1}, x_{d_2}, \ldots, x_{d_\eta}\}$, where $\{x_{d_i}\} \in X'_d$.

Note that, in Algorithm 4.1, $\hat{G}_d$ and G''_d are considered nondeterministic automata, *i.e.*, $\hat{f}_d : \hat{X}_d \times \Sigma_o \to 2^{\hat{X}_d}$ and $f''_d : X''_d \times \Sigma_o \to 2^{X''_d}$. However, Algorithm 4.1 can also be used if $\hat{G}_d$ is deterministic by considering the codomain of $\hat{f}_d$ equal to the set of all singletons formed with the elements of $\hat{X}_d$.

Theorem 4.1 $L(G'_d) \subseteq L(G''_d)$.

Proof. Let $G_p = (X_p, \Sigma, f_p, x_0)$ depicted in Figure 4.4 be an automaton that models a given DES, where $\Sigma = \{\sigma_1, \sigma_2, \sigma_3, \ldots, \sigma_{n-2}, \sigma_{n-1}, \sigma_n\}$. The proof of this theorem is straightforward by construction of G'_p. Note that $L(G_p) = \overline{\{\sigma_1\sigma_3\sigma_5 \ldots \sigma_{n-1}\sigma_n, \sigma_2\sigma_4\sigma_6 \ldots \sigma_{n-2}\sigma_n\}}$. Let us consider, without loss of generality, the operation of merging states $x_1, x_2 \in X_p$. Automaton G'_p, depicted in Figure 4.5, is the automaton generated after merging states x_1, x_2. Note that when we merge states x_1, x_2 the past behavior of x_1 and x_2 converges to the same state (x_1, x_2). As a consequence, new sequences are created, for instance: $\sigma_1\sigma_4\sigma_6 \ldots \sigma_{n-2}\sigma_n$. Also note that the sequences of $L(G_p)$, $\sigma_1\sigma_3\sigma_5 \ldots \sigma_{n-1}\sigma_n$ and $\sigma_2\sigma_4\sigma_6 \ldots \sigma_{n-2}\sigma_n$, belongs to $L(G'_p)$. Thus, $L(G'_p) =$

Algorithm 4.1: MERGE($\hat{G}_d$,x,y)

Input: $\hat{G}_d = (\hat{X}_d, \Sigma_o, \hat{f}_d, \hat{x}_{0_d})$, $x, y \in \hat{X}_d$
Output: $G''_d = (X''_d, \Sigma_o, f''_d, x''_{0_d})$

1 $X''_d = (\hat{X}_d \setminus \{x, y\}) \cup \{x \cup y\}$
2 **if** $\hat{x}_{0_d} \in \{x, y\}$ **then**
3 $\quad x''_{0_d} \leftarrow x \cup y$
4 **else**
5 $\quad x''_{0_d} \leftarrow \hat{x}_{0_d}$
6 Let $xy = x \cup y$. Define transition function f''_d, for all $\sigma \in \Sigma_o$, as:

(i) $f''_d(xy, \sigma) = \hat{f}_d(x, \sigma) \cup \hat{f}_d(y, \sigma)$, if $\left(\hat{f}_d(x, \sigma) \cup \hat{f}_d(y, \sigma)\right) \cap \{x, y\} \neq \emptyset$

(ii) $f''_d(xy, \sigma) = \{xy\} \cup (\hat{f}_d(x, \sigma) \setminus \{x, y\}) \cup (\hat{f}_d(y, \sigma) \setminus \{x, y\})$, if $\hat{f}_d(x, \sigma) \cap \{x, y\} \neq \emptyset$ or $\hat{f}_d(y, \sigma) \cap \{x, y\} \neq \emptyset$

(iii) $f''_d(z, \sigma) = \hat{f}_d(z, \sigma)$, for all $z \in \hat{X}_d \setminus \{x, y\}$, if $\hat{f}_d(z, \sigma) \cap \{x, y\} = \emptyset$

(iv) $f''_d(z, \sigma) = \{xy\} \cup (\hat{f}_d(z, \sigma) \setminus \{x, y\})$, for all $z \in \hat{X}_d \setminus \{x, y\}$, if $\hat{f}_d(z, \sigma) \cap \{x, y\} \neq \emptyset$

$\overline{\{\sigma_1\sigma_3\sigma_5 \ldots \sigma_{n-1}\sigma_n, \sigma_1\sigma_4\sigma_6 \ldots \sigma_{n-2}\sigma_n, \sigma_2\sigma_3\sigma_5 \ldots \sigma_{n-1}\sigma_n, \sigma_2\sigma_4\sigma_6 \ldots \sigma_{n-2}\sigma_n\}}$. Clearly, $L(G_p) \subseteq L(G'_p)$, which completes the proof. ■

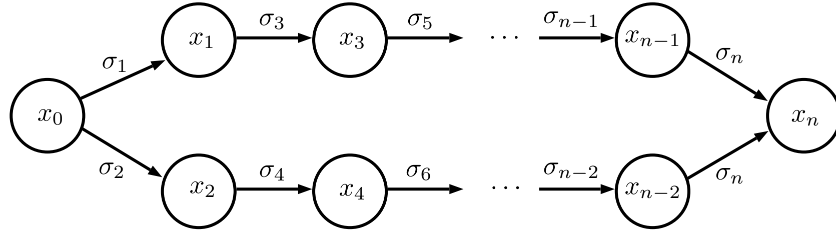


Figure 4.4: Automaton G_p.

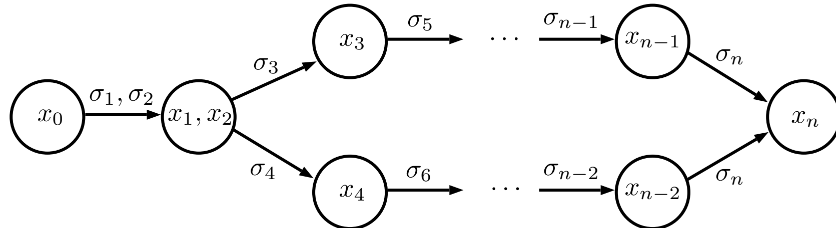


Figure 4.5: Automaton G'_p.

According to Theorem 4.1, the language generated by G''_d, obtained by merging states of G'_d, contains the language generated by G'_d. This language growth must not

alter the diagnosability of the fault event σ_f, and cannot change the delay bound for diagnosis. In order to do so, the following property must hold.

Property 4.1

(*i*) *For all* $s \in L \backslash L_N$*, such that* $P_o(s) \notin P_o(L_N)$*, we have that* $f''_d(x''_{0_d}, P_o(s)) = \{\{F\}\}$.

(*ii*) *For all* $\omega \in L_N$*,* $f''_d(x''_{0_d}, P_o(\omega)) \cap \{\{F\}\} = \emptyset$.

Property 4.1 (*i*) guarantees that all faulty sequences $s \in L \setminus L_N$, that can be diagnosed by G_d, are also diagnosed using G''_d and with the same delay. In addition, Property 4.1 (*ii*) avoids the generation of false positives.

In order to obtain G''_d, only negative and uncertain states of G'_d are merged. Thus, it is necessary to define which states of G'_d can be merged to guarantee Property 4.1.

Definition 4.1 *States* $x, y \in X'_d \setminus \{\{F\}\}$ *are said to be contradictory if there exists* $\sigma \in \Sigma_o$ *such that* $f'_d(x, \sigma) = \{F\}$*, and* $f'_d(y, \sigma) \subset X_N \cup X_{NY}$*, or vice-versa. Otherwise, states* x *and* y *are said to be noncontradictory.*

Note, according to Definition 4.1, that x is noncontradictory with itself, for all $x \in X'_d \setminus \{\{F\}\}$.

The next theorem presents a necessary condition for guaranteeing that G''_d satisfies Property 1.

Theorem 4.2 *If Property 4.1 holds, then contradictory states of* G'_d *are not merged to obtain* G''_d*.*

Proof. Let us consider, without loss of generality, two contradictory states $x, y \in X'_d \setminus \{\{F\}\}$, *i.e.*, there exists $\sigma \in \Sigma_o$ such that $f'_d(x, \sigma) = \{F\}$ and $f'_d(y, \sigma) \subset X_N \cup X_{NY}$. Let $\tilde{s} \in L$ be the sequence with the greatest length such that $f'_d(x_{0_d}, P_o(\tilde{s})) = x$. Thus, according to the computation of G'_d, we have that $s = \tilde{s}\sigma \in L \setminus L_N$ and $P_o(s) \notin P_o(L_N)$. If states x and y are merged, forming a unique state $xy = x \cup y$, then, according to line 6 of Algorithm 4.1, $f''_d(x''_{0_d}, P_o(\tilde{s})) = \{xy\}$, and $f''_d(x''_{0_d}, P_o(\tilde{s})\sigma) = \{f'_d(x, \sigma)\} \cup \{f'_d(y, \sigma)\}$, if $f'_d(y, \sigma) \notin \{x, y\}$, or $f''_d(x''_{0_d}, P_o(\tilde{s})\sigma) = \{f'_d(x, \sigma)\} \cup \{xy\}$, if $f'_d(y, \sigma) \in \{x, y\}$. Thus, since by assumption $f'_d(y, \sigma) \subset X_N \cup X_{NY}$, G''_d is uncertain about the fault occurrence after the observation of $P_o(\tilde{s})\sigma$, Property 4.1 is violated. ■

Note that not merging contradictory states is only a necessary condition to satisfy Property 4.1, *i.e.*, it is possible that merging noncontradictory states also violates Property 4.1. For example, when there exists a sequence of events $s \in \Sigma_o^*$ of length

greater than one such that $f'_d(x,s) = \{F\}$ and $f'_d(y,s) \subset X_N \cup X_{NY}$ or vice-versa, and x and y are merged, then Property 4.1 does not hold. This leads to the following definition.

Definition 4.2 *States $x, y \in X'_d \setminus \{\{F\}\}$ are said to be possibly mergeable if there does not exist a sequence of events $s \in \Sigma_o^*$ such that $f'_d(x,s) = \{F\}$ and $f'_d(y,s) \subset X_N \cup X_{NY}$, or vice-versa.*

Theorem 4.3 *Two states $x, y \in X'_d \setminus \{\{F\}\}$ can be merged to form G''_d satisfying Property 1, only if x, y are possibly mergeable.*

Proof: Let us consider, without loss of generality, four states $x, y, w, z \in X'_d \setminus \{\{F\}\}$ such that x, y are noncontradictory states, *i.e.*, there exists $\sigma \in \Sigma_o$ such that $f'_d(x,\sigma) \subset X_N \cup X_{NY}$ and $f'_d(y,\sigma) \subset X_N \cup X_{NY}$, and w, z are contradictory states, *i.e.*, there exists $\sigma' \in \Sigma_o$ such that $f'_d(w,\sigma') = \{F\}$ and $f'_d(z,\sigma') \subset X_N \cup X_{NY}$. Consider also that $f'_d(x,\sigma) = w$ and $f'_d(y,\sigma) = z$.

Let $\tilde{s} \in L$ be the sequence with the greatest length such that $f'_d(x_{0_d}, P_o(\tilde{s})\sigma) = w$ Thus, according to the computation of G'_d, we have that $s = \tilde{s}\sigma\sigma' \in L \setminus L_N$ and $P_o(s) \notin P_o(L_N)$. If states x and y are merged, forming a unique state $xy = x \cup y$, then, according to line 6 of Algorithm 4.1, $f''_d(x''_{0_d}, P_o(\tilde{s})) = \{w\}$, and $f''_d(x''_{0_d}, P_o(\tilde{s})\sigma') = w$ and, at the same time, $f''_d(x''_{0_d}, P_o(\tilde{s})\sigma') = z$, which leads us to conclude that G''_d is nondeterministic. Thus, since by assumption $f'_d(z,\sigma) \subset X_N \cup X_{NY}$, there exists a sequence of events $s \in \Sigma_o^*$ such that $f'_d(x,s) = \{F\}$ and $f'_d(y,s) \subset X_N \cup X_{NY}$, and G''_d is uncertain about the fault ocurrence after the observation of $P_o(\tilde{s})\sigma'$, violating Property 4.1. Therefore, x, y cannot be merged to form G''_d satisfying Property 4.1 and, consequently, are not possibly mergeable. ■

Note that the condition presented in Theorem 4.3 is also only necessary, since in Definition 4.2 it is not taken into account the exceeding language that can be generated after merging x and y. This case is illustrated in the following example.

Example 4.2 *Consider the system modeled by automaton G, shown in Figure 4.6, where $\Sigma = \{a, b, c, d, \sigma_f\}$, $\Sigma_o = \{a, b, c, d\}$ and $\Sigma_{uo} = \Sigma_f = \{\sigma_f\}$, and its diagnoser G'_d depicted in Figure 4.7. According to Definition 4.2, it is possible to verify that states $\{3N\}$ and $\{4N, 6Y\}$ are possibly mergeable. If we compute G''_d by merging these two states, we obtain the automaton depicted in Figure 4.8. Note that now the observed sequence $s = ca$ leads to both states $\{2N\}$ and $\{F\}$, which shows that the fault is no longer detected after the observation of sequence $s' = abcca$ as in G'_d. Therefore, although $\{3N\}$ and $\{4N, 6Y\}$ are possibly mergeable, they cannot be merged without affecting the diagnosability of the system language.*

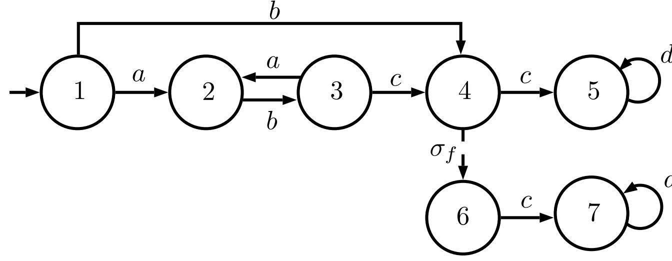


Figure 4.6: Automaton G of Example 4.2.

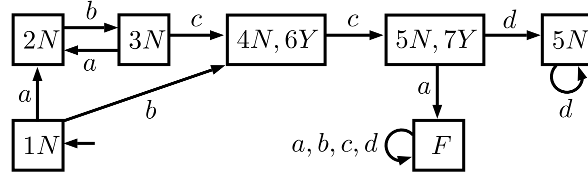


Figure 4.7: Automaton G'_d of Example 4.2.

4.3 Computation of Not Mergeable States

In previous section, we defined that the diagnoser reduction procedure is made through merging two states, but stated that two contradictory states could not be merged. Then, it is necessary to identify, for each state, the set of states that cannot be merged with this state.

In Algorithm 4.2 we present a method for computing all pairs of states of automaton G'_d that are certainly not mergeable according to Definition 4.2. The pairs of states that are certainly not mergeable are stored in set M_{not}. This algorithm is presented as follows.

Note, according to Definition 4.2, that contradictory states are not mergeable. Thus, in line 2 of Algorithm 4.2, C is added to M_{not}. In addition, note that if a pair of states (w, z) leads, after the occurrence of an event $\sigma_o \in \Sigma_o$, to a non-mergeable pair of states, then (w, z) is also non-mergeable. Thus, in lines 7 and 8, these states are added to M_{not}. A *first in, first out* queue Q is used in Algorithm 4.2, where $head[Q]$, $Enqueue(Q, (w, z))$, and $Dequeue(Q)$, denote, respectively, the procedures

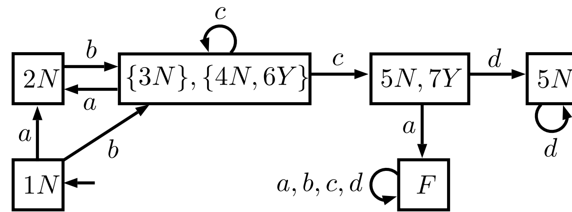


Figure 4.8: Nondeterministic automaton G''_d of Example 4.2.

Algorithm 4.2: NOT_MERGEABLE(G'_d)

Input: $G'_d = (X'_d, \Sigma_o, f'_d, x'_{0_d})$
Output: Set M_{not} formed of all pairs of states that are not mergeable

1 $C := \{(x, y) \in X'_d \setminus \{\{F\}\} \times X'_d \setminus \{\{F\}\} : x, y$ are contradictory$\}$
2 $M_{not} \leftarrow C$
3 Flag all pairs of M_{not}
4 Form a first in, first out queue Q with the pairs of states of C in any order
5 **while** $Q \neq \emptyset$ **do**
6 $\quad (x, y) \leftarrow head[Q]$
7 $\quad$ **for** *each* $(w, z) \in X'_d \setminus \{\{F\}\} \times X'_d \setminus \{\{F\}\}$ *not flagged such that there exists* $\sigma \in \Sigma_o$ *satisfying* $f'_d(w, \sigma) = x$ *and* $f'_d(z, \sigma) = y$ **do**
8 $\quad\quad M_{not} \leftarrow M_{not} \cup \{(w, z)\}$
9 $\quad\quad$ Flag (w, z)
10 $\quad\quad Enqueue(Q, (w, z))$
11 $\quad Dequeue(Q)$

for obtaining the first element of Q, adding (w, z) to the end of Q, and removing the first element of Q.

Since each pair of states is flagged only once in Algorithm 4.2, then each pair is enqueued and dequeued only once. Thus, the complexity of Algorithm 4.2 is $O(|X'_d|^2)$, which is associated with the number of pairs of states of G'_d.

Example 4.3 *Consider diagnoser G'_d depicted in Figure 4.3, where $\Sigma = \{a, b, c, d, t, \sigma_f\}$, $\Sigma_o = \{a, b, c, d, t\}$ and $\Sigma_{uo} = \Sigma_f = \{\sigma_f\}$, and suppose we want to compute all pairs of states of G'_d that are not mergeable according to Definition 4.2. Thus, we need to compute M_{not} using Algorithm 4.2. In line 1 of Algorithm 4.2, we form set C of all pairs of contradictory states of G'_d according to Definition 4.1:*

$$C = \{(\{3N\}, \{4N, 7Y\}), (\{3N\}, \{5N, 8Y\}), (\{4N, 7Y\}, \{6N\}), (\{5N, 8Y\}, \{6N\})\}.$$

In line 2, C is added to M_{not}, and, in line 4, a first in, first out queue Q is formed with the pairs of states of C. After that, the main procedure in the while loop of line 5 begins. If it is not possible to reach a pair of states (x, y) in M_{not} from a pair of states (w, z) after the occurrence of an event $\sigma \in \Sigma_o$, then (x, y) is removed from Q. Otherwise, the pair (w, z) is added to M_{not}. Let us consider that the first element of queue Q is $(\{3N\}, \{4N, 7Y\})$. From Figure 4.3, it can be seen that pair $(\{3N\}, \{4N, 7Y\})$ is reached by pair $(\{1N\}, \{6N\})$ with event t. Thus, $(\{1N\}, \{6N\})$ is added to M_{not}, but $(\{3N\}, \{4N, 7Y\})$ is not removed from Q, since this state is also reached from another pair of states. The algorithm continues and finds that pair $(\{3N\}, \{4N, 7Y\})$ is reached by pair $(\{1N\}, \{3N\})$ with event t. Thus,

$(\{1N\}, \{3N\})$ *is also added to* M_{not}*. Since there does not exist another pair of states that reaches state* $(\{3N\}, \{4N, 7Y\})$*, then it is removed from* Q*. The algorithm continues until* Q *becomes empty. Then, the set* M_{not} *obtained is*

$$M_{not} = \{(\{1N\}, \{3N\}), (\{1N\}, \{6N\}), (\{3N\}, \{4N, 7Y\}), (\{3N\}, \{5N, 8Y\}), (\{4N, 7Y\}, \{6N\}), (\{5N, 8Y\}, \{6N\})\}.$$

In next section, we present an algorithm for computation of a deterministic diagnoser. In this work we only compute deterministic diagnosers.

4.4 Computation of Deterministic Diagnoser

It is important to remark that the diagnoser automaton is, in general, deterministic since the objective is to track the observed sequence of events generated by the plant and, from the reached states of the diagnoser, identifies the fault occurrence. Thus, two possibilities for obtaining the deterministic reduced diagnoser G_d^r are possible: (i) to merge states of G'_d obtaining possibly a nondeterministic automaton, and then to determinize it; or (ii) to merge states of G'_d in such a way that the reduced automaton is already deterministic. Since the determinization may lead to an exponential growth of the deterministic automaton with respect to the number of states of the nondeterministic one, then we adopt approach (ii) to obtain directly a deterministic automaton G_d^r.

In Algorithm 4.3, we present a method to compute a deterministic diagnoser G''_d obtained after merging two states $x, y \in \hat{X}_d$ of an automaton $\hat{G}_d$. In order to do so, Algorithm 4.1 is executed until the transition function $\tilde{f}_d$ becomes deterministic, *i.e.*, only one state is possible to be reached after the occurrence of a feasible event. Thus, G''_d obtained using Algorithm 4.3 is deterministic. Another important characteristic of G''_d is that, as in [2] and [10], its state set X'_d is formed of pairwise disjoint sets.

Algorithm 4.3: DET_MERGE$(\hat{G}_d, x, y)$

Input: $\hat{G}_d = (\hat{X}_d, \Sigma_o, \hat{f}_d, \hat{x}_{0_d})$, and $x, y \in \hat{X}_d$
Output: $G''_d = (X''_d, \Sigma_o, f''_d, x''_{0_d})$

1 $\tilde{G}_d \leftarrow$ MERGE$(\hat{G}_d, x, y)$
2 **while** $\exists w \in \tilde{X}_d$ *such that* $|\tilde{f}_d(w, \sigma)| > 1$*, for* $\sigma \in \Sigma_o$ **do**
3 $\quad$ $\tilde{G}_d \leftarrow$ MERGE$(\tilde{G}_d, \tilde{x}, \tilde{y})$, where $\tilde{x}, \tilde{y} \in \tilde{f}_d(w, \sigma)$
4 $\hat{G}_d \leftarrow \tilde{G}_d$

The complexity of merging states is $O(|\hat{X}_d| \times |\Sigma|)$, and in the worst case the while loop in algorithm 4.3 merge pair of states until there is only one state, *i.e.*, there

will be $|\hat{X}_d| - 1$ merges. Thus, the complexity of Algorithm 4.3 is $O(|\hat{X}_d|^2 \times |\Sigma|)$.

Note that, in order to guarantee the determinism of automaton, after merging two states x, y, it is necessary to merge all pairs of states that are reachable from x and y after the occurrence of the same observable sequence. Thus, it is possible that the state merging of two possibly mergeable states x, y leads to the state merging of two not mergeable states, which shows that x, y cannot be merged. In addition, it is possible that merging two states (x, y), forces the state merging of another pair of states that contains x or y. Let us consider, without loss of generality, that the state reached from (x, y) after the occurrence of an observable event, is (x, z). In this case, in order to guarantee the determinism, it is necessary to merge the three states x, y, and z into a single one. In order to do so, all pairs of states formed with x, y, and z must be mergeable. Since the condition of Theorem 4.3 to guarantee Property 4.1 is only necessary, we need to state a necessary and sufficient condition for verifying Property 4.1. Let $MS(x, y)$ denote the set formed of the sets of states of G'_d that must be merged to guarantee the determinism of the reduced diagnoser, after merging states $x, y \in X'_d \setminus \{\{F\}\}$. Then, the following necessary and sufficient condition to guarantee that states $x, y \in X'_d$ are mergeable can be stated.

Theorem 4.4 *States $x, y \in X'_d$ are mergeable if, and only if, $F \notin M$ for all $M \in MS(x, y)$.*

Proof: ($\Rightarrow$) Let G''_d be the deterministic diagnoser obtained by merging states x, y of G'_d according to Algorithm 4.3. According to the construction of G''_d, it can be seen that if $s \in L(G'_d)$ is such that $f'_d(x'_{0_d}, s) = x'_d$, then $x'_d \subseteq x''_d$, where $x''_d = f''_d(x''_{0_d}, s)$. Thus, if $F \notin M$, for all $M \in MS(x, y)$, then all faulty sequences that leads G'_d to $\{F\}$, also leads G''_d to state $\{F\}$, *i.e.*, all faulty sequences that can be diagnosed using G'_d can also be diagnosed using G''_d. Using the same reasoning it can be seen that all fault-free sequences do not lead G''_d to state $\{F\}$, which shows that G''_d does not raise false alarms.

($\Leftarrow$) Let us suppose now that $F \in M$, where $M \in MS(x, y)$. Then, there exists a state $x_d \in X_N \cup X_{NY}$, such that $x_d \in M$. According to the construction of G''_d, there are different sequences $s_1, s_2 \in L(G'_d)$, such that $f'_d(x'_{0_d}, s_1) = \{x_d\}$ and $f'_d(x'_{0_d}, s_2) = \{F\}$. Thus, if state M of G''_d is reached, we are uncertain about which sequence s_1 or s_2 has occurred, and, consequently, we are uncertain about the occurrence of the fault event. Thus, the diagnosis of the fault event, if possible, will be delayed if sequence s_2 is observed. ■

In next section, we present the algorithm for computing the set $MS(x, y)$ for each pair of states of diagnoser.

4.5 Computation of Mergeable States Set

Considering that we desire a deterministic reduced diagnoser, it is necessary to guarantee the pairwise disjointness between each state set at each step of the merging procedure. Then, in Algorithm 4.4, we present a method to compute all sets $MS(x, y)$, formed only of sets of mergeable states of a deterministic automaton $\hat{G}_d$, obtained after merging states $x, y \in \hat{X}_d$. The sets $MS(x, y)$ are stored in a list ML. Thus, if x, y are not mergeable then $MS(x, y)$ is not added to list ML. Since, in Algorithm 4.4, the necessary condition presented in Theorem 4.3 is used without computing the complete automaton G''_d obtained using Algorithm 4.3, then the computational cost for computing $MS(x, y)$ can be reduced.

In Algorithm 4.4, in lines 8 and 9, the states (w, z) reached from state (x, y) are computed, and the mergeability of w, z is tested in lines 10 to 12 using the necessary condition provided in Theorem 4.3. Note that this condition can be easily verified, and, if it does not hold true, then x, y is not mergeable and the algorithm can continue checking the mergeability of another pair of states. If (w, z) is possibly mergeable, in line 14, it is verified if there exists in $MS(x, y)$ a set of states with at least one element equal to w or z. If this set does not exist, then, in line 15, (w, z) is added to $MS(x, y)$. On the other hand, if there is a set $M \in MS(x, y)$ such that w or z belongs to M, then, in lines 17 to 26, it is verified if it is possible to replace M with the new merged set in $MS(x, y)$. In order to do so, all pairs of states formed with one element from $\{w, z\}$ and the other from M must be possibly mergeable. Finally, in lines 29 and 30, the necessary and sufficient condition of Theorem 4.4 is verified, and if (x, y) are mergeable, then $[(x, y), MS(x, y)]$ is added to list ML.

Since, at each run of the while loop of Algorithm 4.4, pairs of states are verified for each pair of states $(x, y) \in \hat{X}_d \times \hat{X}_d$, then the computational complexity of Algorithm 4.4 is $O(|\hat{X}_d|^4 \times |\Sigma|)$.

Example 4.4 *Consider the plant automaton G depicted in Figure 4.1, and its diagnoser G'_d, depicted in Figure 4.3, whose set of contradictory states M_{not} has been computed in Example 4.3. In order to obtain the list of mergeable states ML, it is necessary to compute $MS(x, y)$ for each pair of states $(x, y) \in (X'_d \times X'_d) \setminus M_{not}$, according to Algorithm 4.4. In line 4, the first pair (x, y) is added to $MS(x, y)$ and in line 5 a first in-first out queue Q is formed with pair (x, y). In line 6 the while loop begins.*

Consider that the first state in Q is $(\{1N\}, \{2N\})$. Note that pair $(\{1N\}, \{2N\})$ does not reach any pair of states of G'_d with the same event. Since $(\{1N\}, \{2N\})$ is mergeable, then set $(\{1N\}, \{2N\})$ is added to $MS(\{1N\}, \{2N\})$.

Let $(\{3N\}, \{6N\})$ be the next state added to Q in line 5. Then, pair $(\{2N\}, \{3N\})$ that is reached from $(\{3N\}, \{6N\})$ with event d is computed in line

Algorithm 4.4: COMPUTE_MS($\hat{G}_d$)

Input: $\hat{G}_d = (\hat{X}_d, \Sigma_o, \hat{f}_d, \hat{x}_{0_d})$
Output: List ML where each element is of the form $[(x, y), MS(x, y)]$

```
1   M̂_not ← NOT_MERGEABLE(Ĝ_d)
2   ML ← NULL
3   for each pair of states (x, y) ∈ (X̂_d × X̂_d) \ M̂_not do
4       MS(x, y) ← {{x, y}}
5       Form a first in-first out Q with pair (x, y)
6       while Q ≠ ∅ do
7           (x, y) ← head[Q]
8           for each σ ∈ Γ_Ĝd(x) ∩ Γ_Ĝd(y) do
9               (w, z) ← (f̂_d(x, σ), f̂_d(y, σ))
10              if (w, z) ∈ M̂_not then
11                  MS(x, y) ← ∅
12                  go to line 2
13              else
14                  if ∄M ∈ MS(x, y) such that {w, z} ∩ M ≠ ∅ then
15                      MS(x, y) ← MS(x, y) ∪ {{w, z}}
16                  else
17                      M̂ ← ∅
18                      for each set M ∈ MS(x, y) such that {w, z} ∩ M ≠ ∅ do
19                          U = {(u, v) : u ∈ {w, z} and v ∈ M}
20                          if U ∩ M̂_not ≠ ∅ then
21                              MS(x, y) ← ∅
22                              go to line 2
23                          else
24                              M̂ ← M̂ ∪ M
25                              MS(x, y) ← MS(x, y) \ M
26                      MS(x, y) ← MS(x, y) ∪ {M̂ ∪ {w, z}}
27              Enqueue(Q, (w, z))
28          Dequeue(Q)
29      if ∄M ∈ MS(x, y) such that F ∈ M then
30          Add [(x, y), MS(x, y)] to list ML
```

9, and the mergeability of $(\{2N\},\{3N\})$ *is tested in line 10. Since* $(\{2N\},\{3N\}) \notin M_{not}$*, then, it is verified in line 14 if* $\{\{2N\},\{3N\}\} \cap \{\{3N\},\{6N\}\} = \emptyset$*. Since* $\{\{2N\},\{3N\}\} \cap \{\{3N\},\{6N\}\} \neq \emptyset$*, then, in line 19, set* $U = \{(\{2N\},\{6N\})\}$ *is formed, and in line 20 it is verified if* $(\{2N\},\{6N\})$ *is not mergeable. Since* $(\{2N\},\{6N\})$ *is mergeable, then set* $\{\{2N\},\{3N\},\{6N\}\}$ *is added to* $MS(\{3N\},\{6N\})$*.*

Algorithm 4.4 continues until all mergeable pairs x, y *whose merging leads to a deterministic diagnoser are added to* ML*.* $MS(x,y)$ *for each pair of mergeable states are presented in the sequel.*

$$
\begin{aligned}
MS(\{1N\},\{2N\}) &= \{\{1N\},\{2N\}\},\\
MS(\{1N\},\{4N,7Y\}) &= \{\{\{1N\},\{4N,7Y\}\}\},\\
MS(\{1N\},\{5N,8Y\}) &= \{\{\{1N\},\{5N,8Y\}\}\},\\
MS(\{2N\},\{3N\}) &= \{\{2N\},\{3N\}\},\\
MS(\{2N\},\{4N,7Y\}) &= \{\{\{2N\},\{4N,7Y\}\}\},\\
MS(\{2N\},\{5N,8Y\}) &= \{\{\{2N\},\{5N,8Y\}\}\},\\
MS(\{2N\},\{6N\}) &= \{\{2N\},\{6N\}\},\\
MS(\{3N\},\{6N\}) &= \{\{2N\},\{3N\},\{6N\}\},\\
MS(\{4N,7Y\},\{5N,8Y\}) &= \{\{\{4N,7Y\},\{5N,8Y\}\}\}.
\end{aligned}
$$

After merging a pair of states $x, y \in X'_d$ of G'_d, all state merging defined in $MS(x,y)$ must be performed. Then, a new automaton G''_d is computed. Thus, as shown in Theorem 4.1, $L(G'_d) \subseteq L(G''_d)$, it is possible that two states $x'', y'' \in X''_d$ are not mergeable even if all pairs of states formed with one element from x'' and the other element from y'' are mergeable, as illustrated in the following example.

Example 4.5 *Consider the system modeled by automaton* G*, shown in Figure 4.9, where* $\Sigma = \{a,b,c,\sigma_f\}$*,* $\Sigma_o = \{a,b,c\}$ *and* $\Sigma_{uo} = \Sigma_f = \{\sigma_f\}$*. First, we compute diagnoser automaton* G_d*, depicted in Figure 4.10. Since* G_d *does not have indeterminate cycles, then* L *is diagnosable with respect to* P_o *and* σ_f*. In this case, state* $\{6Y\}$ *is the unique positive state of* G_d*. Thus, state* $\{6Y\}$ *can be replaced with* F*, generating the diagnoser automaton* G'_d*, depicted in Figure 4.11. Note that a self-loop is introduced in state* F *labeled with all observable events. Computing* M_{not}*, according to Algorithm 4.2, we obtain:*

$$M_{not} = \{(\{1N\},\{4N,5Y\}),(\{2N\},\{4N,5Y\})\}.$$

After obtaining M_{not} *for* G'_d*, we compute* $MS(x,y)$ *for each pair of mergeable states*

(x, y), according to Algorithm 4.4, leading to the following sets:

$$\begin{aligned} MS(\{1N\}, \{2N\}) =& \{\{\{1N\}, \{2N\}\}, \{\{4N, 5Y\}, \{3N\}\}\}, \\ MS(\{1N\}, \{3N\}) =& \{\{\{1N\}, \{3N\}\}\}, \\ MS(\{2N\}, \{3N\}) =& \{\{\{2N\}, \{3N\}\}\}, \\ MS(\{3N\}, \{4N, 5Y\}) =& \{\{\{3N\}, \{4N, 5Y\}\}\}. \end{aligned}$$

Note that states $\{1N\}$, $\{2N\}$, and $\{3N\}$ are pairwise mergeable. However, if we merge the three states into a single state $\{\{1N\}, \{2N\}, \{3N\}\}$, we will be forced to merge all states of G'_d into a single state $\{\{1N\}, \{2N\}, \{3N\}, \{4N, 5Y\}, \{F\}\}$, with a self-loop labeled with a, b, c, to guarantee that G''_d is deterministic. Thus, clearly G''_d cannot be used for diagnosis.

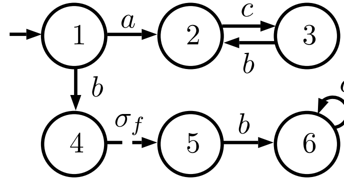


Figure 4.9: Automaton G of Example 4.5.

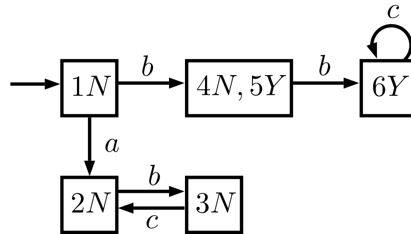


Figure 4.10: Automaton G_d of Example 4.5.

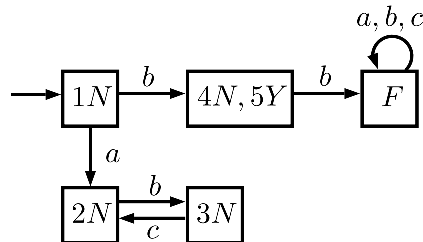


Figure 4.11: Automaton G'_d of Example 4.5.

Example 4.5 shows that after merging a pair of states of G'_d, it is necessary to compute M''_{not} and the sets $MS(x'', y'')$, for all mergeable pairs of states $x'', y'' \in X''_d$

of G''_d, in order to continue the reduction procedure. When no mergeable pair of states is found, then the diagnoser cannot be reduced anymore, and G^r_d has been computed.

Adapting the methods proposed in [2] and [10] to the reduction of diagnosers, states of G'_d are merged according to a predefined order that these states are given. However, the order that the states are merged is not exploited to obtain a reduced automaton with a smaller number of states. In [1], a criterion to choose which states are merged at each step of the reduction procedure is proposed. The basic idea is to find, at each step, the maximal set of states that are pairwise mergeable, and then, merge these states.

In this work, we propose a new criterion for choosing which pair of states, at each step of the reduction procedure, should be merged to generate a reduced diagnoser. We show that the new criterion may generate reduced diagnosers with a smaller number of states than the method proposed in [2] and it will be deterministic, differently from the one obtained using the method proposed in [1].

4.6 Diagnoser Reduction Algorithm

In previous section, we computed the mergeable states set for each state of G'_d. Then, to reduce G'_d as maximum as possible, we establishes a criterion to define in which order the state merging will occur. In this section we propose an algorithm that exactly follows this criterion, and computes the reduced diagnoser G^r_d.

As shown in Example 4.5, the existence of a set formed of pairwise mergeable states is only a necessary condition to merge all states into a single state, since the state merging may alter the mergeability of the other states. Thus, the reduction procedure must be carried out considering the merging of only pairs of states, and, consequently, performing the merging of all states defined in $MS(x, y)$, at each step of the reduction procedure.

Let $G''_d = (X''_d, \Sigma_o, f''_d, x''_{0_d})$ be the diagnoser computed after performing the merging of two states of a diagnoser $\hat{G}_d$. In order to choose which pair of states (x, y) of $\hat{G}_d$ should be merged to obtain G''_d, we define N_m that measures the potential merging of states of G''_d, as follows:

$$N_m = \frac{N_e}{|X''_d|},$$

where N_e is the number of pairs of possibly mergeable states of G''_d. Since the total number of pairs of states of G''_d is given by $\frac{|X''_d|(|X''_d|-1)}{2}$, then

$$N_e = \frac{|X''_d|(|X''_d| - 1)}{2} - |M''_{not}|,$$

where M''_{not} is the set of pairs of states of G''_d that are not mergeable obtained using Algorithm 4.2.

If N_m is large, then there are several possible merging of pairs of states of G''_d, which implies that the chance of merging more states in the next step of the reduction procedure is greater than if N_m is a small number. Using this reasoning, we choose to merge, at each step of the reduction procedure, the pair of states of the diagnoser $\hat{G}_d$ whose merging leads to the diagnoser G''_d with the higher value of N_m. This procedure is described in Algorithm 4.5.

Algorithm 4.5: REDUCED_DIAGNOSER(G'_d)

Input: G'_d
Output: G^r_d

1 $\hat{G}_d \leftarrow G'_d$
2 $ML \leftarrow$COMPUTE_MS($\hat{G}_d, M_{not}$)
3 **while** $ML \neq NULL$ **do**
4 $\quad LN \leftarrow$ NULL
5 $\quad$ **for** *each element* $[(x,y), MS(x,y)]$ *of* ML **do**
6 $\quad\quad G''_d \leftarrow$ DET_MERGE($\hat{G}_d, x, y$)
7 $\quad\quad M''_{not} \leftarrow$ NOT_MERGEABLE(G''_d)
8 $\quad\quad$ Compute $N_m = \frac{\frac{|X''_d|(|X''_d|-1)}{2} - |M''_{not}|}{|X''_d|}$
9 $\quad\quad$ Add $[N_m, G''_d]$ to list LN
10 $\quad$ Find the maximum number N_{max} of N_m in list LN
11 $\quad$ Choose an automaton $G''_{d_{max}}$ associated with N_{max} in LN
12 $\quad \hat{G}_d \leftarrow G''_{d_{max}}$
13 $\quad ML \leftarrow$COMPUTE_MS($\hat{G}_d, M_{not}$)
14 $G^r_d \leftarrow \hat{G}_d$

Following the steps of Algorithm 4.5, the reduced diagnoser G^r_d is computed from G'_d. In lines 2 and 3, M_{not} and ML are computed for automaton $\hat{G}_d$. Then, while there are mergeable states in ML, in line 7, automaton G''_d is computed for each pair $[(x,y), MS(x,y)]$ of ML, and, in line 9, N_m is computed for each G''_d. In line 10, the maximum number N_{max} of N_m is computed. Note that more than one automaton G''_d can have the same value of N_m. Thus, in line 11, an automaton $G''_{d_{max}}$ associated with N_{max} is chosen. After that, in line 13, ML is computed for G''_d. If ML is NULL, *i.e.*, there is no mergeable pair of states, the algorithm stops and returns $G^r_d = G''_d$. On the other hand, the algorithm continues, and a new reduction is carried out.

In the worst case, the while loop in Algorithm 4.5 merge pair of states until there is only one state, *i.e.*, there will be $|X'_d| - 1$ merges. Also, at each run of the for loop of Algorithm 4.5, pairs of states are verified for each pair of states $(x,y) \in X'_d \times X'_d$, and ML is computed. Hence, the complexity of Algorithm 4.5 is $O(|X'_d|^5)|$. The

diagnoser reduction procedure is illustrated in the following example.

Example 4.6 *Let us consider again the plant automaton G, and diagnoser G'_d, both respectively depicted in Figures 4.1 and 4.3 of Example 4.4, and suppose we want to obtain a reduced diagnoser, G^r_d, that is capable of verify diagnosability of language L and has the same diagnosis delay of G'_d. Then, it is necessary to compute Algorithm 4.5. In line 2, the set M_{not} is computed, whose result was obtained in Example 4.3, and is shown in the sequel.*

$$\begin{aligned} M_{not} = &\{(\{1N\},\{3N\}),(\{1N\},\{6N\}),(\{3N\},\{4N,7Y\}),(\{3N\},\{5N,8Y\}),\\ &(\{4N,7Y\},\{6N\}),(\{5N,8Y\},\{6N\})\}. \end{aligned}$$

In line 3, the $MS(x,y)$ for each pair of mergeable states is computed, whose results were obtained in Example 4.4, and are shown in the sequel.

$$\begin{aligned} MS(\{1N\},\{2N\}) =& \{\{1N\},\{2N\}\},\\ MS(\{1N\},\{4N,7Y\}) =& \{\{\{1N\},\{4N,7Y\}\}\},\\ MS(\{1N\},\{5N,8Y\}) =& \{\{\{1N\},\{5N,8Y\}\}\},\\ MS(\{2N\},\{3N\}) =& \{\{2N\},\{3N\}\},\\ MS(\{2N\},\{4N,7Y\}) =& \{\{\{2N\},\{4N,7Y\}\}\},\\ MS(\{2N\},\{5N,8Y\}) =& \{\{\{2N\},\{5N,8Y\}\}\},\\ MS(\{2N\},\{6N\}) =& \{\{2N\},\{6N\}\},\\ MS(\{3N\},\{6N\}) =& \{\{2N\},\{3N\},\{6N\}\},\\ MS(\{4N,7Y\},\{5N,8Y\}) =& \{\{\{4N,7Y\},\{5N,8Y\}\}\}. \end{aligned}$$

Then, the algorithm computes N_m for each $MS(x,y)$ obtained in previous step, and adds the result for each pair (x,y) to list LN. G'_d has six states, then $|X'_d| = 6$. Thus, when a pair is merged, G''_d has five states, i.e., $|X''_d| = 5$.

Let us compute N_e for pair $(\{1N\},\{2N\})$. Note that when states $\{1N\}$ and $\{2N\}$ are merged, the new M''_{not} set can be computed as

$$\begin{aligned} M''_{not} = &\{((\{1N\},\{2N\}),\{3N\}),((\{1N\},\{2N\}),\{6N\}),(\{3N\},\{4N,7Y\}),\\ &(\{3N\},\{5N,8Y\}),(\{3N\},\{6N\}),(\{4N,7Y\},\{6N\}),\\ &(\{5N,8Y\},\{6N\})\}. \end{aligned}$$

The computation of N_e and N_m are, respectively, given by: $N_e = (5\times(5-1))/2-7 = 3$, and $N_m = N_e/|X''_d| = 3/5$. After computing N_e for each pair of states, we compute N_m. The list LN that contains the results of N_m for each pair of states is

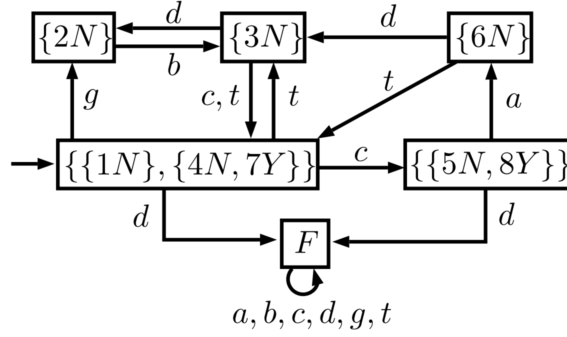


Figure 4.12: Merging of states $\{1N\}$ and $\{4N, 7Y\}$ of G'_d of Example 4.6.

presented in the sequel.

$$LN = \{[3/5, (\{1N\}, \{2N\})], [6/5, (\{1N\}, \{4N, 7Y\})], [6/5, (\{1N\}, \{5N, 8Y\})],\\ [4/5, (\{2N\}, \{3N\})], [3/5, (\{2N\}, \{4N, 7Y\})], [3/5, (\{2N\}, \{5N, 8Y\})],\\ [4/5, (\{2N\}, \{6N\})], [4/5, (\{3N\}, \{6N\})], [6/5, (\{4N, 7Y\}, \{5N, 8Y\})]\}.$$

Clearly, $N_{max} = 6/5$. Then, according to Algorithm 4.5, any pair of states such that $N_m = 6/5$ can be chose to be merged. For instance, suppose that states $\{1N\}, \{4N, 7Y\}$ are merged. Therefore, the diagnoser G''_d after the first merge of states, depiced in Figure 4.12, is computed.

Since there are mergeable states in G''_d, the while loop restarts, where M_{not} and $MS(x, y)$ are computed for G''_d.

In line 2, the set M_{not} is computed, whose result is shown in the sequel.

$$M_{not} = \{(\{\{1N\}, \{4N, 7Y\}\}, \{3N\}), (\{\{1N\}, \{4N, 7Y\}\}, \{6N\}),\\ (\{3N\}, \{5N, 8Y\}), (\{5N, 8Y\}, \{6N\})\}.$$

In line 3, the $MS(x, y)$ for each pair of mergeable states is computed, whose results are shown in the sequel.

$$MS(\{\{1N\}, \{4N, 7Y\}\}, \{2N\}) = \{\{\{1N\}, \{4N, 7Y\}\}, \{2N\}\},\\ MS(\{\{1N\}, \{4N, 7Y\}\}, \{5N, 8Y\}) = \{\{\{1N\}, \{4N, 7Y\}\}, \{5N, 8Y\}\},\\ MS(\{2N\}, \{3N\}) = \{\{2N\}, \{3N\}\},\\ MS(\{2N\}, \{5N, 8Y\}) = \{\{\{2N\}, \{5N, 8Y\}\}\},\\ MS(\{2N\}, \{6N\}) = \{\{2N\}, \{6N\}\},\\ MS(\{3N\}, \{6N\}) = \{\{2N\}, \{3N\}\{6N\}\}.$$

Then, the algorithm computes N_m for each $MS(x, y)$ obtained in previous step, and adds the result for each pair (x, y) to list LN. G'_d has five states, then $|X'_d| = 5$.

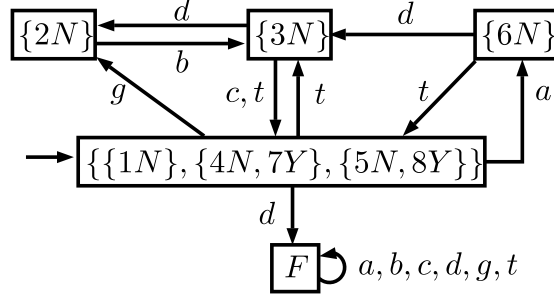


Figure 4.13: Merging of states $\{1N\}$, $\{4N, 7Y\}$ and $\{5N, 8Y\}$ of G'_d of Example 4.6.

Thus, when a pair is merged, G''_d has four states, i.e., $|X''_d| = 4$. The list LN that contains the results of N_m for each pair of states is presented in the sequel.

$$\begin{aligned}LN = &\{[1/4, (\{\{1N\}, \{4N, 7Y\}\}, \{2N\})], [4/4, (\{\{1N\}, \{4N, 7Y\}\}, \{5N, 8Y\})],\\ &[2/4, (\{2N\}, \{3N\})], [1/4, (\{2N\}, \{5N, 8Y\})],\\ &[2/4, (\{2N\}, \{6N\})], [2/4, (\{3N\}, \{6N\})]\}.\end{aligned}$$

Since, $N_{max} = 4/4$, then, pair of states $\{\{1N\}, \{4N, 7Y\}\}, \{\{5N, 8Y\}\}$ is merged, and diagnoser G''_d, depiced in Figure 4.13, is computed.

Continuing Algorithm 4.5, it is verified that G''_d still has mergeable states. Then, the while loop continues, where M_{not} and $MS(x, y)$ are computed for G''_d.

In line 2, the set M_{not} is computed, whose result is shown in the sequel.

$$\begin{aligned}M_{not} = &\{(\{\{1N\}, \{4N, 7Y\}, \{5N, 8Y\}\}, \{3N\}),\\ &(\{\{1N\}, \{4N, 7Y\}, \{5N, 8Y\}\}, \{6N\})\}.\end{aligned}$$

In line 3, the $MS(x, y)$ for each pair of mergeable states is computed, whose results are shown in the sequel.

$$\begin{aligned}MS(\{\{1N\}, \{4N, 7Y\}, \{5N, 8Y\}\}, \{2N\}) = &\{\{\{1N\}, \{4N, 7Y\}, \{5N, 8Y\}\}, \{2N\}\},\\ MS(\{2N\}, \{3N\}) = &\{\{2N\}, \{3N\}\},\\ MS(\{2N\}, \{6N\}) = &\{\{2N\}, \{6N\}\},\\ MS(\{3N\}, \{6N\}) = &\{\{2N\}, \{3N\}\{6N\}\}.\end{aligned}$$

Then, the algorithm computes N_m for each $MS(x, y)$ obtained in previous step, and adds the result for each pair (x, y) to list LN. G'_d has four states, then $|X'_d| = 4$. Thus, when a pair is merged, G''_d has three states, i.e., $|X''_d| = 3$. The list LN that

contains the results of N_m for each pair of states is presented in the sequel.

$$\begin{aligned} LN = \{&[0/3, (\{\{1N\}, \{4N, 7Y\}, \{5N, 8Y\}\}, \{2N\})],\\ &[1/3, (\{2N\}, \{3N\})],\\ &[1/3, (\{2N\}, \{6N\})], [1/3, (\{3N\}, \{6N\})]\}. \end{aligned}$$

Since, $N_{max} = 1/3$, then, any pair of states associated with this value of N_m can be merged. Let us suppose that states $\{3N\}, \{6N\}$ are merged. Since $MS(\{3N\}, \{6N\}) = \{\{2N\}, \{3N\}\{6N\}\}$, the three states $\{2N\}, \{3N\}\{6N\}$ are merged.

Note that it is not possible to reduce G''_d anymore, since $ML = NULL$. Therefore, the reduced diagnoser G^r_d is computed, and it is depicted in Figure 4.14.

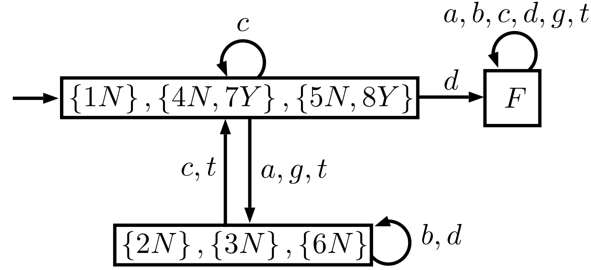


Figure 4.14: Automaton G^r_d of Example 4.6.

Since G^r_d does not have indeterminate cycles, language L is diagnosable with respect to P_o and Σ_f, and its diagnosis delay is $z = 3$. Note that diagnosis delay of original diagnoser G'_d, depicted in Figure 4.3, is also $z = 3$.

It is important to remark that if the reduction method proposed in [1] is used, then the reduced automaton presented in Figure 4.15 is obtained. Also, if the reduction method proposed in [2] is used, then the reduced automaton presented in Figure 4.16 is obtained.

In diagnoser reduction we established that two contradictory states cannot be merged, while in [1] and [2], two states with different control actions cannot be merged, *i.e.*, if an event σ is disabled in state x and $f(y, \sigma)!$, then x, y cannot be merged. Thus, the algorithms proposed in [1] and [2] can be applied to diagnoser reduction according to this adaptation.

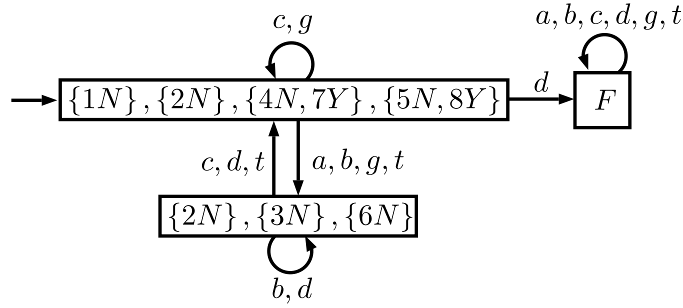


Figure 4.15: Automaton G_d^r of Example 4.6 computed using the method proposed in [1].

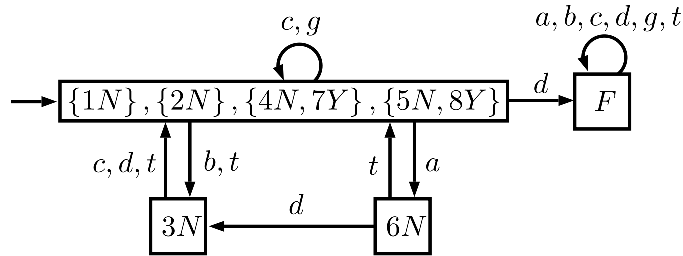


Figure 4.16: Automaton G_d^r of Example 4.6 computed using the method proposed in [2].

Note that the reduced automaton obtained using our method has only three states, while the reduced automaton computed using the method proposed in [2] has four states. Although the reduced diagnoser obtained by method proposed in [1] has three states, it is a nondeterministic diagnoser. This example shows that it is possible, in some cases, to obtain reduced automata with smaller number of states using the proposed strategy in this work than by using the strategy presented in [2], and still obtain a deterministic diagnoser, differently from the method proposed in [1].

Chapter 5

Conclusions and Future Works

Although it has been shown in the literature that the minimization problem of supervisors is NP-hard, empirical results show that it is possible to reduce the diagnoser proposed in [6]. With that in mind, our main goal of this work was to propose an algorithm for the computation of a deterministic reduced diagnoser of DES, where the diagnosability of the system language is preserved and diagnosis delay is the same as the original diagnoser. We show that the algorithm proposed can lead to reduced diagnosers with fewer states than methods proposed in the literature, most notably the supervisor reduction strategies. This suggests the possibility of adapting the method proposed in this work to solve the supervisory reduction problem.

The advantage of this method, and also its difference from the supervisor reduction strategies, is that the criterion to merge states does not depends on the previous order of the states of the automaton model. This allows us to exploit all possible merges for each state, which is the main reason why our method provides better results than [1] and [2] in some cases.

Possible topics of research to continue this work are listed below:

(*i*) Verifying if the method proposed in this work achieves better results than approaches proposed in [1] and [2], when applied to supervisors.

(*ii*) It would be of interest to check if the method proposed here always computes a diagnoser with minimal state size, maintaining the desired properties.

(*iii*) Another future direction is to investigate how to minimize the computational cost of constructing the reduced diagnoser proposed in this work.

(*iv*) Application of diagnoser reduction algorithm to real manufacturing systems.

Bibliography

[1] MINHAS, R. S. *Complexity Reduction in Discrete Event Systems*. Ph.D. thesis, ECE Department, University of Toronto, Canada, 2002.

[2] SU, R., WONHAM, W. M. "Supervisor reduction for discrete-event systems", *Journal of Discrete Event Dynamic Systems*, v. 14, n. 1, pp. 31–53, 2004.

[3] YOO, T.-S., GARCIA, H. E. "Computation of fault detection delay in discrete-event systems". In: *Proceedings of the 14th International Workshop on Principles of Diagnosis, DX'03*, pp. 207–212, Washington, USA, 2003.

[4] YOKOTA, S., YAMAMOTO, T., TAKAI, S. "Computation of the delay bounds and synthesis of diagnosers for decentralized diagnosis with conditional decisions", *Discrete Event Dynamic Syst.*, pp. 1–40, 2016.

[5] TOMOLA, J. H. A., CABRAL, F. G., CARVALHO, L. K., et al. "Robust Disjunctive-Codiagnosability of Discrete-Event Systems Against Permanent Loss of Observations", *IEEE Transactions on Automatic Control*, v. 62, n. 11, pp. 5808–5815, 2017.

[6] SAMPATH, M., SENGUPTA, R., LAFORTUNE, S. "Diagnosability of discrete-event systems", *IEEE Transactions on Automatic Control*, v. 9, n. 40, pp. 1555–1575, 1995.

[7] CLAVIJO, L. B., BASILIO, J. C. "Empirical studies in the size of diagnosers and verifiers for diagnosability analysis", *Discrete Event Dynamic Systems*, v. 27, n. 4, pp. 701–739, 2017.

[8] ZAD, S. H., KWONG, R. H., WONHAM, W. M. "Fault diagnosis in discrete-event systems: Framework and model reduction", *IEEE Transactions on Automatic Control*, v. 48, n. 7, pp. 1199–1212, 2003.

[9] HOPCROFT, J. E., MOTWANI, R., ULLMAN, J. D. *Introduction to Automata Theory, Languages and Computation*. 3 ed. Boston, MA: USA, Addison-Wesley, 2006.

[10] CAI, K., GIUA, A., SEATZU, C. "On Consistent Reduction in Discrete-Event Systems". In: *Proc. 15th IEEE Int. Conf. on Automation Science and Engineering*, pp. 474–479, 2019.

[11] VAZ, A. F., WONHAM, W. M. "On supervisor reduction in discrete-event systems", *International J. Control*, v. 44, n. 2, pp. 475–491, 1986.

[12] CAI, K., WONHAM, W. M. "Supervisor localization: a top-down approach to distributed control of discrete-event systems", *IEEE Transactions on Automatic Control*, v. 3, n. 55, pp. 605–608, 2010.

[13] CASSANDRAS, C., LAFORTUNE, S. *Introduction to Discrete Event Systems.* New York, Inc., Secaucus, NJ, Springer-Verlag, 2008.

[14] RAMADGE, P., WONHAM, W. M. "Supervisory control of a class of discrete event processes", *SIAM J. Control and Optimization*, v. 25, n. 1, pp. 206–230, 1987.

[15] WONHAM, W., CAI, K., RUDIE, K. "Supervisory control of discrete-event systems: A brief history", *Annual Reviews in Control*, v. 45, 2018.

[16] RAMADGE, P., WONHAM, W. M. "The Control of Discrete Event Systems", *Proceedings of the IEEE*, v. 77, n. 1, pp. 81–98, 1989.