COMBINED CENTRALIZED AND DECENTRALIZED FAULT DIAGNOSIS FOR DISCRETE EVENT SYSTEMS

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES
OF
MIDDLE EAST TECHNICAL UNIVERSITY

BY

RUHI KARAV

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR
THE DEGREE OF MASTER OF SCIENCE
IN
ELECTRICAL AND ELECTRONICS ENGINEERING

FEBRUARY 2014

Approval of the thesis:

COMBINED CENTRALIZED AND DECENTRALIZED FAULT DIAGNOSIS FOR DISCRETE EVENT SYSTEMS

submitted by **RUHI KARAV** in partial fulfillment of the requirements for the degree of **Master of Science in Electrical and Electronics Engineering Department, Middle East Technical University** by,

Prof. Dr. Canan Özgen
Dean, Graduate School of **Natural and Applied Sciences** __________

Prof. Dr. Gönül Turhan Sayan
Head of Department, **Electrical and Electronics Eng.** __________

Assoc. Prof. Dr. Şenan Ece Schmidt
Supervisor, **Elect. and Electronics Eng. Dept., METU** __________

Assoc. Prof. Dr. Klaus Werner Schmidt
Co-supervisor, **Mechatronics Eng. Dept., Çankaya Uni.** __________

Examining Committee Members:

Prof. Dr. Semih Bilgen
Electrical and Electronics Engineering Department, METU __________

Assoc. Prof. Dr. Şenan Ece Schmidt
Electrical and Electronics Engineering Department, METU __________

Prof. Dr. Kemal Leblebicioğlu
Electrical and Electronics Engineering Department, METU __________

Assoc. Prof. Dr. Cüneyt Bazlamaçcı
Electrical and Electronics Engineering Department, METU __________

Assist. Prof. Dr. Ulaş Beldek
Mechatronics Engineering Department, Çankaya University __________

Date: __________

ABSTRACT

COMBINED CENTRALIZED AND DECENTRALIZED FAULT DIAGNOSIS FOR DISCRETE EVENT SYSTEMS

Karav, Ruhi
M.S., Department of Electrical and Electronics Engineering
Supervisor : Assoc. Prof. Dr. Şenan Ece Schmidt
Co-Supervisor : Assoc. Prof. Dr. Klaus Werner Schmidt

February 2014, 81 pages

Discrete Event Systems (DES) are used for modeling systems such as manufacturing systems, telecommunication systems and transportation systems. It is possible to incorporate the fault model in the DES model together with a fault diagnosis approach to evaluate the robustness and the reliability of the system at the design stage. There are centralized or decentralized fault diagnosis approaches in the literature. The centralized fault diagnosis achieves stronger results however it does not scale to reasonably large systems because of its complexity. The decentralized diagnosis is applicable to real-life systems with a cost of possible misses of faults.

This thesis proposes a combination of centralized and decentralized fault diagnosis for DES models. To this end, the thesis makes use of the observation that some parts of the faulty DES behavior might be detected by decentralized diagnosis while other parts need a centralized diagnoser. Hence, the overall complexity of the diagnosis is reduced while maintaining the ability to detect all faults. The thesis proposes a systematic diagnosis approach together with the algorithms and practical applications to manufacturing system and communication network examples.

Keywords: Discrete Event System, Centralized Diagnosis, Decentralized Diagnosis, Testing Automaton, Finite Detection Delay, Mask Function, libFAUDES

ÖZ

AYRIK OLAYLI SİSTEMLER İÇİN BİRLEŞİK MERKEZİ VE DAĞITILMIŞ BOZUKLUK TANILAMA

Karav, Ruhi
Yüksek Lisans, Elektrik ve Elektronik Mühendisliği Bölümü
Tez Yöneticisi : Doç. Dr. Şenan Ece Schmidt
Ortak Tez Yöneticisi : Doç. Dr. Klaus Werner Schmidt

Şubat 2014 , 81 sayfa

Günlük yaşamdaki üretim sistemleri, telekomüniskasyon sistemleri ve taşıma sistemleri gibi pekçok alan Ayrık Olaylı Sistem olarak modellenmektedir. Bu ayrık olaylı sistemlerde hatalı olaylar meydana gelebilir. Bu yüzden başka bir alan olan *bozukluk tanılayıcı* incelenmelidir. Literatürde pekçok bozukluk tanılayıcı mimari vardır. Bunlar monolitik, dağıtılmış ve modüler mimarilerdir. Bu çalışmada biz monolitik ve dağıtılmış mimarilere yoğunlaşacağız. Bu iki mimaride de tek bir sistemde hata ayıklama işlemi yapılır. Ancak monolitik mimaride tek ve genel bir bozukluk tanılayıcı kullanılırken, dağıtılmış mimari yerel tanılayıcıları kullanır. Literatürde eğer bir sistem dağıtılmış tanılayıcı mimari ile ayıklanabiliyorsa sadece dağıtılmış mimari kullanılır. Ancak bu sağlamıyorsa monolitik mimari kullanılır. Biz bu iki mimariyi birleştirdik. Bizim mimarimiz mümkün olduğu kadar dağıtılmış, zorunluluk oranında da monolitik kullanmaya dayanmaktadır.

Anahtar Kelimeler: Ayrık Olaylı Sistem, Merkezi Bozukluk Tanılayıcı, Dağıtılmış Bozukluk Tanılayıcı, Test Otomatı, Sınırlı Algılama Gecikmesi, Maske Fonksiyonu, libFAUDES

To my family

ACKNOWLEDGMENTS

I would like to thank my advisors Associate Professors Şenan Ece Schmidt and Klaus Werner Schmidt for their guidance and friendship. They encouraged me to finish thesis work.

I would like to express my appreciation to Mehmet Ali Gülden and Kenan Ahıska for their friendships and for motivating me throughout this work.

There are lots of people that were with me during my thesis work. They always supported me. It is impossible to explain why they are so important. Therefore, i will give names of some of my friends; Ahmet Karakaya, Gökhan Ordu, Alper Avcıoğlu, Serkan Nas, Ömer Batmaz.

My family also provided invaluable support for this work. I would like to thank specially to my wife, Kadriye Sultan for her patience. She always makes me feel loved and cared. I would like to thank my father Davut, my mother Nimet, my brother Enes and my sister Gülendam.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

DES	Discrete Event Systems
G	Generator Automaton
$\overline{H}$	Augmented Specification Automaton
K	Specification Automaton
L	Language
$L^{\texttt{co}}$	Supremal codiagnosable sublanguage
$L^{\texttt{non}}$	Noncodiagnosable sublanguage
M	Global observation mask function
M_1	The first mask function
M_2	The second mask function
T	Testing Automaton
x	The first component of the current state in T
x_0	Initial state of G
x_{next}	The first component of the next state in T
y_0	Initial state of K
y^1	The third component of the current state in T
y^2	The last component of the current state in T
$\overline{y}$	The second component of the current state in T
$\overline{y}_0$	Initial state of $\overline{H}$
$\overline{y}_{next}$	The second component of the next state in T
y^1_{next}	The third component of the next state in T
y^2_{next}	The last component of the next state in T
σ	The first string of the current event in T where $\sigma \in L(G)$
σ_1	The second string of the current event in T where $\sigma_1 \in \Sigma$
σ_2	The third string of the current event in T where $\sigma_2 \in \Sigma$
Σ	Finite event set
Σ_{uo}	Unobservable event set
Σ_o	Observable event set

Σ_f	Faulty event set
ϵ	Empty string
Δ	Set of observation

CHAPTER 1

INTRODUCTION

A large number of technical systems including manufacturing systems [5],[8], telecommunication systems [2], [1], transportation systems [7] or document processing systems [16] can be conveniently modeled as *Discrete Event Systems, DES*. Consequently, the *Failure diagnosis* approaches for DES can be applied to such system models to support robustness and reliability at the design stage.

The fault diagnosis for DES is based on a DES model that includes the faulty system behavior, and a characterization of the possible faults (fault model) in terms of unobservable *failure events* [17] or a *failure specification* [13], [25]. The *diagnosis* goal is to uniquely identify, if an (unobservable) fault occurred, based on the partial observation of the actual system behavior. In this context, a DES is denoted as *diagnosable* if every modeled fault can be detected after a bounded number of event occurrences [17]. In order to analyze if all faults of a DES are diagnosable, the literature provides various approaches that provide diagnosability test for DES [17, 25, 13, 8, 24, 18, 19, 20].

There are different architectures for the fault diagnosis for DES. In the *centralized* architecture, a single diagnosing entity (*diagnoser*), that observes the overall behavior of the system is used [8],[17],[25],[24],[26], [18]. Differently, *decentralized* approaches rely on diagnoser components at different local sites, that communicate among each other or to a coordinating entity, in order to obtain a global diagnosis result [13], [6], [23], [22],[12],[19]. In general, centralized diagnosis provides a stronger diagnosis result, since all observations are obtained and processed by a single entity. However, due to design complexity and due to physical distribution, centralized diagnosis is usually not possible for practical systems. In that case, the usage of a decentralized architecture has to be taken into account.

According to the chosen fault model, the verification of *event-diagnosability* (failure events) and *language-diagnosability*(failure specification) is studied in the literature. In both cases, the diagnosability is investigated separately for the *centralized* and decentralized approaches. In contrast, this thesis proposes to combine decentralized diagnosis and centralized diagnosis. Hereby, the main motivation of this work is the ability to detect faults by decentralized diagnosers whenever possible but to use centralized diagnosis whenever necessary.

Different than the previous works in the literature, this thesis employs the observation that some parts of the faulty DES behavior might be detected by decentralized diagnosis while other parts need a centralized diagnoser. To this end, the thesis is based on the observation from [21] that there is a *supremal codiagnosable sublanguage* that contains the faulty system behavior that can be detected by decentralized diagnosers. The remaining system behavior is then diagnosed using centralized diagnosis.

In view of the described problem setup, the contributions of the thesis are summarized as follows

- We first describe the concept of a supremal codiagnosable sublanguage. This sublanguage contains all strings that can be diagnosed by decentralized diagnosis.
- We next realize an algorithm for the computation of the supremal codiagnosable sublanguage and implement this algorithm in the software library libFAUDES [10, 9]. The algorithm runs in polynomial time.
- We also implement the verification algorithm for codiagnosability according to [13] in the scope of this thesis. Hereby, a mistake in the original formulation of [13] is identified and corrected.
- We demonstrate the applicability of our algorithm by several case studies from manufacturing systems and computer networks.

The remainder of this thesis is organized as follows. Chapter-2 provides the necessary background and notation regarding DES, languages and automata. Chapter-3 includes information about centralized diagnosis and the diagnosability verification algorithm from [25]. In Chapter-4, decentralized diagnosis is described and an algorithm from [13] is implemented in libFAUDES. Chapter-5 includes the definition of the *supremal codiagnosable sublanguage* and its algorithmic computation. In addition, the combination of centralized and decentralized diagnosis is performed in this chapter. Several application examples are provided in Chapter 6 and the last chapter concludes the thesis work.

CHAPTER 2

DISCRETE EVENT SYSTEMS

We first present basic information, notations and definitions about *Discrete Event Systems*, *DES*. The readers can consult [3] for more information. A *system* is considered as a combination of components, whereby the combination of these components can perform a function that is not possible with the individual parts. In the literature, two types of systems are distinguished, *time-driven systems* and *event-driven systems.* In time-driven systems, state transitions occur continuously or with a synchronized clock. On the other hand, in event-driven systems, state changes are dependent on discrete instantaneous actions called *events*. Therefore, an event-driven system is a combination of asynchronous and concurrent event processes. Discrete event systems (DES) are a type of event-driven systems where both the system state space and transitions between system states are discrete. The scope of this thesis work is the consideration of DES.

2.1 Language

In discrete event systems, state transitions occur with actions called *events*. All events in a DES form a finite event set that denoted as an *alphabet* and shown as $\Sigma = \{\sigma_1, \sigma_2, \sigma_3 .... \sigma_n\}$.

2.1.1 String

A sequence of events from alphabet is called as *string*. In this section, some operations and definitions about *string* are given.

$\ast$ **Length of String** : The number of events in a string is called as *length of the string.* Length of s is shown as |s|.

$\ast$ **Empty String** : The string with zero length is called as *empty string*, in other word, there is no event in an empty string. Empty string is written as ϵ.

$\ast$ **Concatenation** : Let x and y be two strings, then *concatenation* of x and y is written as xy. The length of xy is the sum of the length of x and the length of y.

$\ast$ **Suffix and Prefix** : Let w be the concatenation of x and y, $w = xy$, then y is a *suffix* of w and x is a *prefix* of w.

2.1.2 Language

A *regular language* over Σ is a set of finite-length strings from Σ. There are several relevant language operations given below;

$\ast$ **Concatenation** : Let L_a and L_b be two language where L_a,$L_b \subseteq \Sigma^\star$, then L_aL_b=$\{s \in \Sigma^\star : (s = s_as_b)$ for $(s_a \in L_a)$ and $(s_b \in L_b)\}$.

$\ast$ **Kleene Closure** : Let $L \subseteq \Sigma^\star$, then $L^\star = \{\epsilon\} \cup L \cup LL \cup LLL...$ is the set of all possible event sequences that can be formed using the alphabet Σ.

$\ast$ **Empty Language** : The language that does not have any element, denoted by $\emptyset$.

$\ast$ **Prefix Closure** : Let $L \subseteq \Sigma^\star$, then $\overline{L}$=$\{s \in \Sigma^\star : (\exists t \in \Sigma^\star)[st \in L]\}$. L is *prefix-closed* if $L = \overline{L}$.

$*$ **Post-Language** : L/s denotes the post-language of L after s such that;

$$L/s = \{t \in \Sigma^\star | st \in L\} \tag{2.1}$$

$*$ **Natural Projection** : Let observable alphabet $\Sigma_o \subseteq \Sigma$, then P:$\Sigma^\star \to \Sigma_o^\star$ is define by

$$\begin{aligned} &P(\epsilon) = \epsilon \\ &P(\sigma) = \begin{cases} \sigma \text{ if } \sigma \in \Sigma_0 \\ \epsilon \text{ if } \sigma \notin \Sigma_0 \end{cases} \\ &P(s\sigma) = p(s)p(\sigma) \text{ for } s \in \Sigma^\star \text{ and } \sigma \in \Sigma \end{aligned} \tag{2.2}$$

$*$ **Inverse Projection** : P_L^{-1} is defined as

$$P_L^{-1}(y) = \{s \in L : P(s) = y\} \tag{2.3}$$

2.2 Automata

Automata are used to model discrete event systems. Deterministic Automata and Non-deterministic automata are the types of automata considered in this thesis.

2.2.1 Deterministic Automata

A deterministic automaton is given by a five-tuple, $G = (X, \Sigma, \gamma, x_0, X_m)$ where,

$$\begin{aligned} &X \text{ is the set of states.} \\ &\Sigma \text{ denoted alphabet.} \\ &\gamma : X \times \Sigma \to X \text{ is the transition function.} \\ &x_0 \text{ shows initial state.} \\ &X_m \text{ is marked states where } X_m \in X. \end{aligned} \tag{2.4}$$

The most important properties of deterministic automata are, there is only one initial state and for every predecessor state and event set, there is at most one successor state defined by the transition function.

2.2.2 Non-deterministic Automata

Non-deterministic automaton is also five-tuple, $G = (X, \Sigma, \gamma, x_0, X_m)$ where,

$$\begin{aligned} &X \text{ is the set of states.} \\ &\Sigma \cup \{\epsilon\} \text{ denoted alphabet with empty string.} \\ &\gamma : X \times \Sigma \rightarrow 2^X \text{ is the transition function.} \\ &X_0 \text{ is the set of initial states.} \\ &X_m \text{ is marked states where } X_m \in X. \end{aligned} \tag{2.5}$$

In this type of automaton, transitions from the same predecessor state and with the same event to multiple successor states are allowed. In addition, there can be multiple initial states.

2.2.3 Operations on Automata

∗ **Accessible Part:** There may be some states not reachable from initial state in an automaton. After deleting these unnecessary states of automaton A, we have *Ac(A)*, accessible part of automaton A. If A=*Ac(A)*, then A is accessible.

∗ **Coaccessible Part:** q state of an automaton A is coaccessible if it is possible to reach a marked state from this q state. If we delete all states that are not coaccessible, we have *CoAc(A)*, coaccessible part of A. If A=*CoAc(A)*, then A is coaccessible.

∗ **Trim:** An automaton which is both accessible and coaccessible, is trim.

$$Trim(A) = Ac(CoAc(A)) = CoAc(Ac(A)) \tag{2.6}$$

∗ **Synchronous Composition:** Let $A = (X, \Sigma_A, \gamma_A, x_0, X_m)$ and $B = (Y, \Sigma_B, \gamma_B, y_0, Y_m)$, then synchronous composition of A and B,

$A||B = Ac((X \times Y, \Sigma_A \cup \Sigma_B, \gamma, x_0.y_0, X_m \times Y_m))$ where

$$\gamma(q_A.q_B, \sigma) = \begin{cases} \gamma_A(q_A, \sigma).\gamma_B(q_B, \sigma) \text{ if } \gamma_A(q_A, \sigma) \text{ and } \gamma_B(q_B, \sigma) \text{ are defined.} \\ \gamma_A(q_A, \sigma).q_B \text{ if } \gamma_A(q_A, \sigma) \text{ is defined but } \gamma_B(q_B, \sigma) \text{ is not.} \\ q_A.\gamma_B(q_B, \sigma) \text{ if } \gamma_B(q_B, \sigma) \text{ is defined but } \gamma_A(q_A, \sigma) \text{ is not.} \\ q_A.q_B \text{ if } \gamma_A(q_A, \sigma) \text{ and } \gamma_B(q_B, \sigma) \text{ are not defined.} \end{cases} \tag{2.7}$$

∗ **Deadlock:** If there is deadlock in an automaton, then there is not any possible event for at least one state in this automaton.

CHAPTER 3

CENTRALIZED DIAGNOSIS

The main scope of diagnosability is detecting the occurrence of unobservable failure events in discrete event systems. In this chapter, the *centralized fault diagnosis* of discrete event system is studied. First the definition of diagnosability is given. Then an algorithm to check diagnosability is described.

3.1 Diagnosability

As defined in [17], [25] and [14], a DES is modeled by an automaton $G = (X, \Sigma, \delta, x_0)$ where X is the state space, Σ is the alphabet, δ is the transition function and x_0 is the initial state of the system. [17] introduces two types of events in Σ, *observable events* Σ_o and *unobservable events* Σ_{uo}. The definition of these event types is based on the general modeling of DES. On the one hand, there are events whose occurrence can be directly seen such as actuators events that are generated by DES controllers or sensor events that are generated from sensor signals. Such events are termed observable. On the other hand, there can be events that happen in a DES but whose occurrence cannot be directly seen such as faults. These events are considered unobservable.

$$\Sigma = \Sigma_o \cup \Sigma_{uo} \tag{3.1}$$

Mask Function where M:$\Sigma \rightarrow \Delta \cup \{\epsilon\}$ is the key element of diagnosability. It is derived from natural projection described in section-2.1.2. Observable events are detected by sensor part,in other words each observable event is recorded as sensor readings by sensor. This sensor readings are called observation. Unobservable events can not be detected by sensor, so the observation of these events is equal to ϵ. Therefore, mask function maps each event $\sigma \in \Sigma$ to its observation $\Delta \cup \{\epsilon\}$ where Δ is the *set of observations*. The event types can be described by using mask function. An event is observable when its observation is not equal to ϵ, and an event is unobservable when its observation is equal to ϵ. Mask function can be generalized to strings as;

$$M(s\sigma) = \begin{cases} M(s)M(\sigma) \text{ if } \sigma \in \Sigma_o \\ M(s) \text{ if } \sigma \in \Sigma_{uo} \end{cases} \tag{3.2}$$

System faults are formulated as a*specification language*, $K = \overline{K} \subseteq L(G)$. Specification language consists of observable and unobservable events, and every string that belongs to the specification language is considered as a non-faulty string of the system. On the other hand, a string s is faulty if $s \in L(G) \setminus K$.

It is desired to detect faulty strings $s \in L(G) \setminus K$ in G by observation of the system behavior through the observation mask M. As in [17], let G model a DES, let K be specification language and let $M : \Sigma \rightarrow \Delta \cup \{\epsilon\}$ be an observation mask. Then, K is *language-diagnosable* with respect to G and M if

$$\begin{aligned} &(\exists n \in N)(\forall s \in L(G) \setminus K)(\forall st \in L(G), |t| \geq n \text{ or } st \text{ deadlocks}) \\ &\Rightarrow (\forall u \in M^{-1}M(st) \cap L(G), u \notin K) \text{ where } N \text{ is the set of natural numbers} \end{aligned} \tag{3.3}$$

This equation remarks that a language is diagnosable if it satisfies two conditions. First condition is that faulty and non-faulty strings do no share the same observation result after a bounded number of event occurrences after a fault occurrence or if a deadlock state is reached. This means that, for any extension st of s that is longer than a given bound n, there should not be an $u \in K$ such that $M(u) = M(st)$.

In this part, we show three different application. In the first application generator including deadlock state and loops, is diagnosable. Plant in the second

application has deadlock state and is not diagnosable. In the final application generator includes indeterminate cycles and is not diagnosable.
Let G in figure-3.5 be plant, K in figure-3.6 be specification automaton and M in table-3.1 be global observation mask where a and c are faulty events. This plant is diagnosable with respect to K and M, because c event can be diagnosed after g event and b event can be diagnosed if e event occurs between d and f events in string.
In the second application, G in figure-3.10 is not diagnosable with respect to specification automaton K in figure-3.11 and global observation mask M given in table-3.2 where f event is faulty. Because observable event e can occur in both faulty and nonfaulty string.
In the last application, let the automaton in figure-3.8 be the plant automaton G, and the automaton in figure-3.6 be the specification automaton K and M be observation mask in table-3.1 where $\Sigma_{uo} = \{a, b\}$ and $\Sigma_f = \{a\}$. It is said that *L(G)* is not diagnosable with respect to K and M, because both faulty string and nonfaulty string may consist of $d(ef)^\star$ events.

3.2 Testing Diagnosability

In [25], Yoo and Garcia suggest an algorithm to check diagnosability of a language. This algorithm, depends on the construction of a *weighted directed graph*. This graph is shown as $W(G, K, M) = (V(G, K), E(G, K, M))$ where $G = (X, \Sigma, \delta, x_0)$ is plant automaton, $C = (Y, \Sigma, \nu, y_0)$ is specification automaton with $K = L(C)$, M is observation mask. The automaton part in weighted directed graph is built as,

$$\begin{aligned} &V \subseteq \{Y \times Y \times X \times \{normal, confused\}\} \cup \{Block\} \\ &\nu(y_1, \sigma_1) = y_1', \nu(y_2, \sigma_2) = y_2' \text{ and } \delta(x, \sigma_2) = x'. \end{aligned} \tag{3.4}$$

The edge function, $E(G, K, M)$, can get two values. The values are "0" and "-1". The rules of edge function is given in the below. Some rules are illustrated in figures. These figures show parts of the weighted directed graph generated from the automaton G in figure-3.5, specification $K = L(C)$ in figure-3.6 and the mask function from table-3.1.

$*$ where σ_1 and $\sigma_2 \in L(G)$ and $M(\sigma_1) = M(\sigma_2) = \epsilon$

1. $(y_1, y_2, x, normal) \rightarrow (y_1', y_2, x, normal)$ and $E = 0$ if y_1' is defined.
2. $(y_1, y_2, x, normal) \rightarrow (y_1, y_2', x', normal)$ and $E = 0$ if y_2' and x' are defined.
3. $(y_1, y_2, x, normal) \rightarrow (y_1, y_2, x', confused)$ and $E = -1$ if y_2' is not defined but x' is defined.

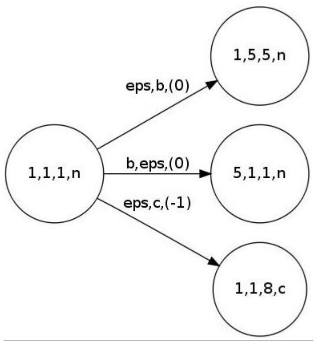

Figure 3.1: The first three rules of Weighted Directed Graph W

4. $(y_1, y_2, x, confused) \rightarrow (y_1', y_2, x, confused)$ and $E = 0$ if y_1' is defined.

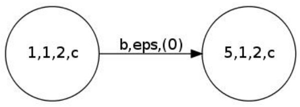

Figure 3.2: The fourth rule of Weighted Directed Graph W

5. $(y_1, y_2, x, confused) \rightarrow (y_1, y_2, x', confused)$ and $E = -1$ if x' is defined.

$*$ where σ_1 and $\sigma_2 \in L(G)$ and $M(\sigma_1) = M(\sigma_2) \neq \epsilon$

1. $(y_1, y_2, x, normal) \rightarrow (y_1', y_2', x', normal)$ and $E = 0$ if y_1',y_2' and x' are defined.
2. $(y_1, y_2, x, normal) \rightarrow (y_1', y_2, x', normal)$ and $E = -1$ if y_1' and x' are defined but y_2' is not defined.

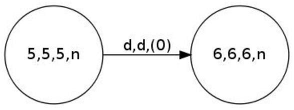

Figure 3.3: The sixth rule of Weighted Directed Graph W

3. $(y_1, y_2, x, confused) \rightarrow (y_1', y_2, x', confused)$ and $E = -1$ if y_1' and x' are defined.

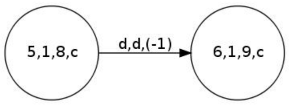

Figure 3.4: The eighth rule of Weighted Directed Graph W

4. $(y_1, y_2, x, confused) \rightarrow Block$ and $E = 0$ if x' is not defined.

The automaton, V consists of three components. The first two components, y_1 and y_2, are states from K and the last component, x, is a state from G. All transition of V include two events. The first event belongs to the specification language *L(K)* and is applied to y_1, and the other event belongs to *L(G)* and is applied to y_2 and x.
y_2 and x change simultaneously as long as second applied event is also an element of *L(K)* and indicator becomes *normal*. When there is no y_2' state defined in C, the indicator part changes to $confused$. Therefore, the indicator *confused, normal* shows whether strings s leading to the respective verifier state are non-faulty ($s \in L(K)$) or faulty ($L(G) \setminus L(K)$).

Yoo and Garcia prove that after constructing weighted directed graph, loops with negative edge in W indicate the violation of diagnosability. That is, if there is any loop with negative edge in weighted directed graph, then the language is not diagnosable with respect to specification language and mask function.

Table3.1: Global Observation Mask of The Plant

$\sigma \in \Sigma$	a	b	c	d	e	f	g
$M(\sigma)$	ϵ	ϵ	ϵ	D	E	F	G

We illustrate the algorithm on an example. Let G in figure-3.5 be a plant automaton, C in figure-3.6 be specification automaton with $K = L(C)$ and M be observation mask in table-3.1. These observation means that a, b and c are unobservable events with respect to given mask function. Moreover, it can be argued that a fault occurs whenever a or c happen.

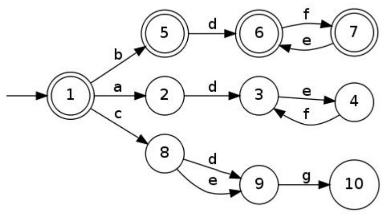

Figure 3.5: Generator, G

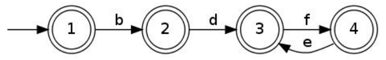

Figure 3.6: Specification Automaton, K

The weighted directed graph *V(G,K,M)* is shown in figure-3.7. In this figure, weight functions are shown in the parenthesis for each transition and ϵ is shown as *eps*. The language *L(G)* is diagnosable with respect to K and M, since there is not a loop, having negative weight edge, or a block vertex.

Let the automaton in figure-3.8 be the plant automaton G, and the automaton in figure-3.6 be the specification automaton C and M be observation mask in table-3.1 where $\Sigma_{uo} = \{a, b\}$ and $\Sigma_f = \{a\}$. It is said that *L(G)* is not diagnosable with respect to K and M, because there is a loop with negative weight edge. The loop between *(6,1,3,c)* and *(7,1,4,c)* can be seen in figure-3.9.

In the final application, let G in figure-3.10 be generator, K in figure-3.11 be specification and M in table-3.2 be global observation mask where f is the faulty event. G generator is not diagnosable with respect to K specification and M mask function where f event is faulty, because there is a *BLOCK* state in weighted directed graph shown in figure-3.12.

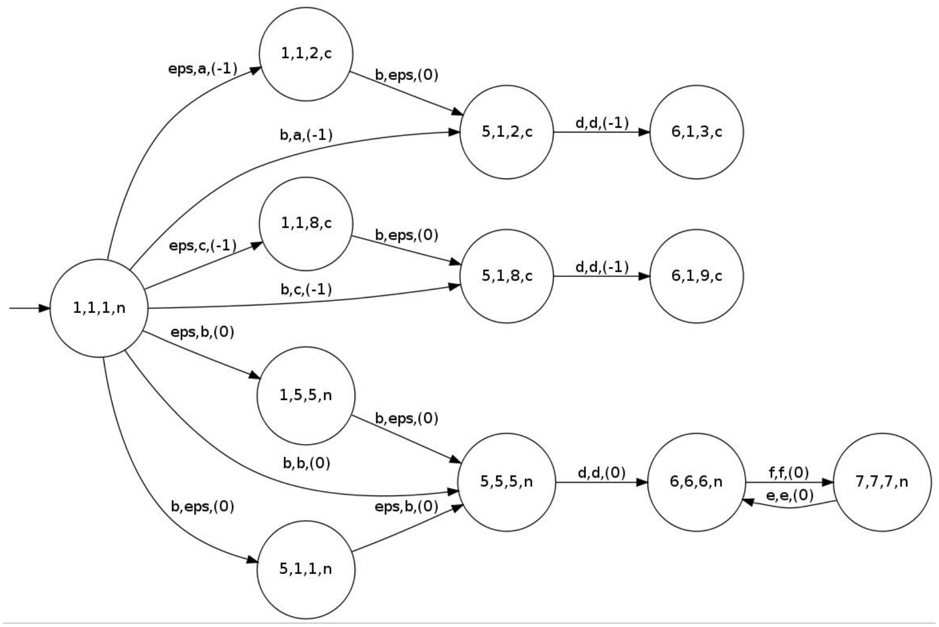

Figure 3.7: Weighted Directed Graph W

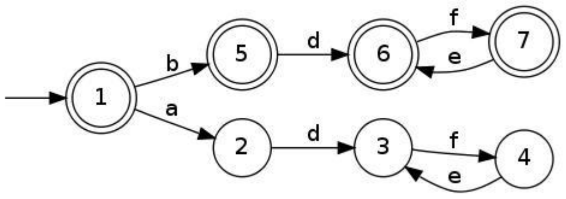

Figure 3.8: Generator, G_2

3.3 Complexity Analysis

State of weighted directed graph consists of three states and an edge function. The first two states are from specification automaton, and the last state is an element of generator function. Therefore, complexity of constructing weighted directed graph is equal to $O(|Y| \times | > | \times |X|)$ where $|Y|$ is the number of states in specification automaton and $|X|$ is the number of states in plant automaton. However, complexity of searching negative edge in weighted directed graph is more complicated. This problem can be solved by *Bellman-Ford Algorithm* approach. As stated in [4], Bellman-Ford algorithm computes the shortest paths from initial state in a weighted graph. In other words, this algorithm finds

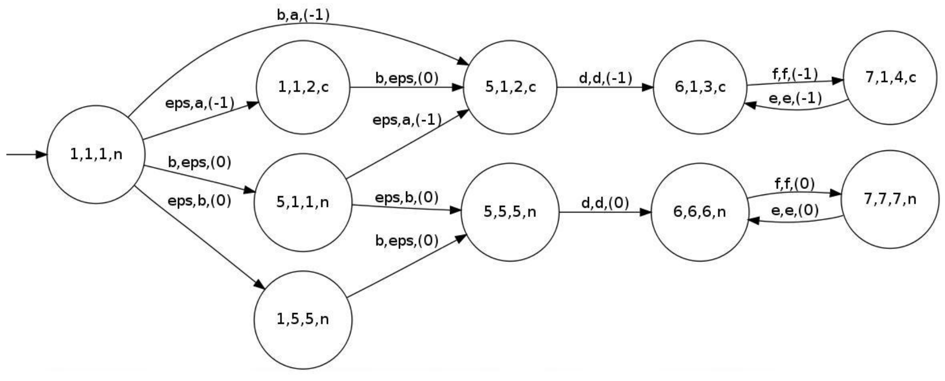

Figure 3.9: Weighted Directed Graph with Negative Edge Weight

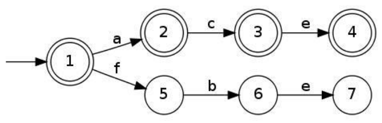

Figure 3.10: Generator

negative edge cycles. Complexity of Bellman-Ford Algorithm is $O(|V||E|)$ where $|V|$ is the number of vertices and $|E|$ is the number of edges.

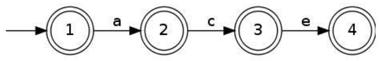

Figure 3.11: Specification

Table3.2: Global Observation Mask of The Plant

$\sigma \in \Sigma$	a	b	c	e	f
$M(\sigma)$	ϵ	ϵ	ϵ	E	ϵ

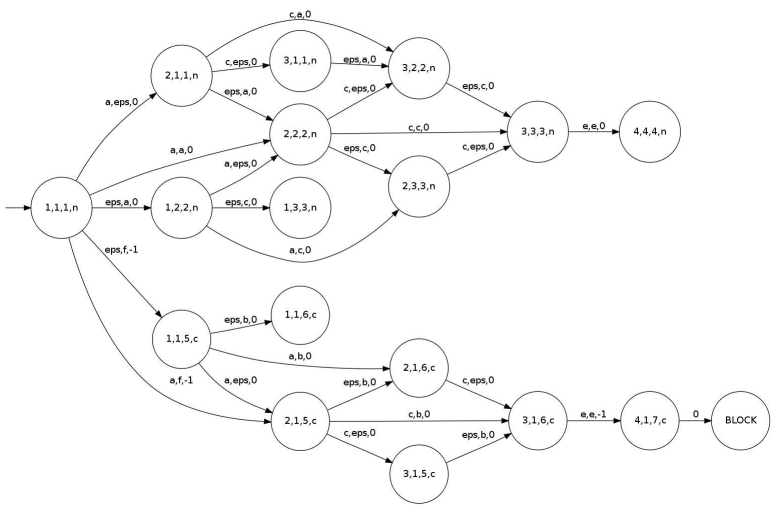

Figure 3.12: Weighted Directed Graph including BLOCK state

CHAPTER 4

DECENTRALIZED DIAGNOSIS

In the previous chapter, a global observation mask for the diagnosability of entire generator is discussed. This algorithm can be useful for small systems, but in large systems this algorithm becomes infeasible. The decentralized diagnosis architecture is developed for this type of systems. Failure events in a global system are diagnosed by more than one local observation masks in decentralized diagnosis architecture. The architecture is symbolized in figure-4.1.

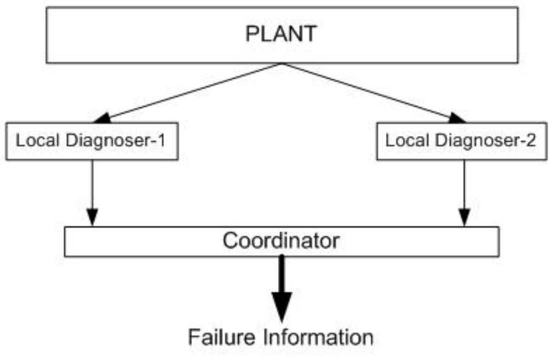

Figure 4.1: Decentralized Diagnosis Architecture

In this thesis work, we focus on the algorithm suggested by Qui and Kumar in [13]. In this chapter firstly notation of codiagnosability is examined. Then codiagnosability checking algorithm is given. Finally this algorithm is implemented in the libFAUDES software library for discrete event systems.

4.1 Co-diagnosability

Let $G = (X, \Sigma, \delta, x_0)$ be a plant automaton, K be a prefix-closed specification language and assume there are m decentralized observation mask, $M_i : \Sigma \to \Delta_i \cup \{\epsilon\}$ where $i \in \{1, 2, 3....m\}$. *(G,K)* is *codiagnosable* with respect to M_i if

$$\begin{aligned}&(\exists n \in N)(\forall s \in L(G) \setminus K)\\ &(\forall st \in L(G) \setminus K, |t| \geq n \text{ or } st \text{ deadlocks})\\ &\Rightarrow (\exists i \in \{1,2,3,....,m\})(\forall u \in M_i^{-1} M_i(st) \cap L(G), u \in L(G) \setminus K).\end{aligned} \tag{4.1}$$

This equation remarks that there are two conditions that should be fulfilled. Firstly, for all arbitrary long faulty strings, there should be at least one mask M_i that can distinguish the string from non-faulty strings. Secondly, for faulty deadlock strings, there should not be any non-faulty string that generates the same observation under all observation masks.

In order to develop the codiagnosability verification algorithm, the condition of non-codiagnosability is presented. (G, K) is not codiagnosable with respect to M_i if

$$\begin{aligned}&(\forall n \in N)(\exists s \in L(G) \setminus K)\\ &(\exists st \in L(G) \setminus K, |t| \geq n \text{ or } st \text{ deadlocks})\text{such that}\\ &(\forall i \in \{1,2,3,....,m\})(\exists u_i \in M_i^{-1} M_i(st) \cap L(G), u_i \in K).\end{aligned} \tag{4.2}$$

The condition in (4.1) has two cases, deadlocking strings or strings with finite detection delay. For a simpler algorithmic treatment, in [13] Qui and Kumar suggest a trick such that only one case has to be considered. If there exists a deadlocking state, a self-loop with the event ϵ is added. Hence the system becomes deadlock free, whereby adding ϵ does not change the generated observations. Let t be a deadlocking string, and the observation result of this string be $M_i(t)$, then if ϵ is inserted to that string, observation result becomes $M_i(t\epsilon^\star)$ and

$$M_i(t) = M_i(t\epsilon^{\star}) \tag{4.3}$$

For the automaton shown in figure-4.2, if the system is on state-6, then this system deadlocks. Because there is not any transition from state-6. However, if we add self-loop to state-6 in figure-4.3, then the system becomes deadlock-free. In figure-4.3 ϵ event is denoted by *eps*.

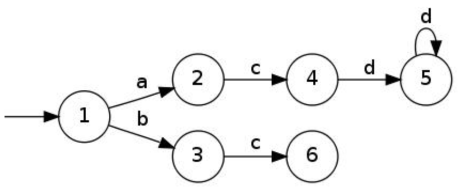

Figure 4.2: A Plant without self loop

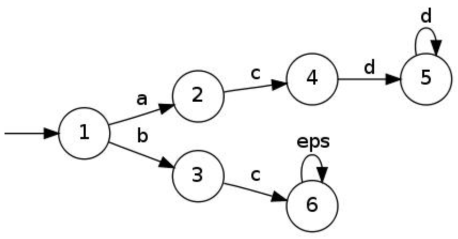

Figure 4.3: A Plant with self loop

Considering the described trick, it is possible to assume that only plants without deadlocks are given. Hence, the codiagnosability verification algorithm is developed for such plants in the sequel. Plants with deadlocks are extended by ϵ-selfloops and can then be analyzed by the same algorithm.

In this part, we check codiagnosability of three different cases. In the first application, the language generated by G, shown in figure-4.17 is codiagnosable with respect to specification automaton K in figure-4.18 and observation masks given in table-4.2 where a and f are unobservable events.Because, the first local observation mask can diagnose b and c events and if there is b event before c event in string then this string is not faulty otherwise this string is faulty. Moreover, the second local observation mask can diagnose c and d events. If there is d event in string then the string is faulty.

In the second example, the language generated by G, shown in figure-4.20 includes cycle and this automaton is not codiagnosable with respect to specification automaton K in figure-4.21 and observation masks function given in table-4.3 where s and f are unobservable events. Because, the first observation mask can see $ABD^{\star}$ for both faulty and nonfaulty strings and the second observation mask can see $ACE^{\star}$ for both faulty and nonfaulty strings. Therefore, faulty and nonfaulty strings can not be diagnosed by using these local observation masks. In the last application, generator G and specification K has deadlocking states. Let G be automaton given in figure-5.18, K be automaton shown in figure-5.19 and local observation masks be given in table-5.2 where a and f are unobservable events and f is faulty event. This generator is not codiagnosable, because b event is observable with the first local observation mask and this event can occur in both faulty and nonfaulty strings. Moreover, c event is observable with the second local observation mask and it can not be diagnosed by the second observation mask.

4.2 Codiagnosability Checking

In this section, the algorithm for checking codiagnosability as proposed by Qui and Kumar in [13] is presented. The algorithm is based on the plant automaton $G = (X, \Sigma, \delta, x_0)$, the specification automaton $H = (Y, \Sigma, \beta, y_0)$ with $L(H) = K$ and the observation masks M_i, $i = 1, 2$. Note that we describe the algorithm for the case of two local sites for simplicity. The extension to more than two local sites is straightforward. The algorithm consists of three steps.

4.2.1 Constructing the Augmented Specification Automaton, $(\overline{H})$

The first step of the co-diagnosability checking algorithm is the construction of the *Augmented Specification Automaton* denoted by $\overline{H} = (\overline{Y}, \Sigma, \overline{\beta}, y_0)$. This automaton is derived from the specification automaton H. The state set of the *augmented specification automaton* consists of the state set of the *specification automaton* and a new state, F. F is reached in $\overline{H}$ whenever the specification K

is violated.

$$\overline{Y} = Y \cup \{F\} \tag{4.4}$$

The transition function of $\bar{H}$ is derived from the transition function of H by keeping all transitions of H and adding transitions to the state F whenever an event is not defined at a state of H. Formally, the transition function of the augmented specification automaton is stated as follows. For all $\overline{y} \in \overline{Y}$ and $\sigma \in \Sigma$

$$\overline{\beta}(\overline{y}, \sigma) = \begin{cases} \beta(\overline{y}, \sigma) \text{ if } [\overline{y} \in Y] \wedge [\beta(\overline{y}, \sigma) \text{ exists}] \\ \text{F if } [\overline{y} = F] \vee [\beta(\overline{y}, \sigma) \text{ does not exist}] \end{cases} \tag{4.5}$$

4.2.2 Constructing the Testing Automaton

The second step of the co-diagnosability checking algorithm is constructing a *testing automaton* $T = (Z, \Sigma^{\mathtt{T}}, \gamma, z_0)$. This automaton includes information about the plant automaton, augmented specification automaton and the observation masks. The testing automaton is constructed according to the following rules,

$$\begin{aligned} & Z = X \times \overline{Y} \times Y \times Y \\ & z_0 = (x_0, \overline{y_0}, y_0, y_0) \\ & \Sigma^{\mathtt{T}} = (\Sigma \cup \{\epsilon\}) \times (\Sigma \cup \{\epsilon\}) \times (\Sigma \cup \{\epsilon\}) \\ & \gamma = Z \times \overline{\Sigma}^{3} \rightarrow \text{Z is defined as} \\ & \forall z = (x, \overline{y}, y^1, y^2) \in Z, \\ & \sigma^{\mathtt{T}} = (\sigma, \sigma^1, \sigma^2) \in \Sigma^{\mathtt{T}} \setminus \{(\epsilon, \epsilon, \epsilon)\} \\ & \gamma(z, \sigma^{\mathtt{T}}) = (\alpha(x, \sigma), \overline{\beta}(\overline{y}, \sigma), \beta(y^1, \sigma^1), \beta(y^2, \sigma^2)) \end{aligned} \tag{4.6}$$

if and only if

$$[M_1(\sigma) = M_1(\sigma^1), M_2(\sigma) = M_2(\sigma^2)] \wedge [\alpha(x,\sigma), \beta(y^1,\sigma^1), \beta(y^2,\sigma^2) \neq \emptyset]. \tag{4.7}$$

That is, the state space of the testing automaton is $X \times \overline{Y} \times Y \times Y$. In other words, the first component of each testing automaton state belongs to the plant automaton G, the second component belongs to the augmented specification automaton $\overline{H}$ and the last two components are from the specification automaton K.

For each transition of T, a tuple with three events from $(\Sigma, \Sigma_1, \Sigma_2)$ is applied to the respective predecessor states. The first event is executed in G and is applied to the first and second components of testing automaton. The second and third events are executed in H, whereby the second event is applied to the third component of the testing automaton and the third event is applied to the last component of the testing automaton. These three events should be related such that the observation of the first and second event should be equal with respect to M_1 and the observations of the first and third event should be equal with respect to M_2. Moreover, all three events can not be ϵ at the same time.

The second component of the testing automaton state keeps track of possible faulty strings. Whenever a faulty string is followed in the plant, this second component moves to the state label F and remains there. The third component of the testing automaton state follows strings in the specification that have the same observation as the plant string under the observation mask M_1, whereas the fourth component of the testing automaton state follows strings in the specification that have the same observation as the plant string under the observation mask M_2. That is, if a state with second component F is reached in the testing automaton, this implies that there is a faulty plant string that is confused with a non-faulty plant string with the same observation under both observation masks. Hence, such faulty string cannot be diagnosed.

4.2.3 Checking the Violation of Codiagnosability

A language is said to be codiagnosable with respect to the specification language, if the occurrence of faulty strings can be detected by at least one local observation mask after a bounded number of event occurrences. The codiagnosability check can be done by searching *indeterminate cycle* in the testing automaton. Hereby, a cycle is given by

$$\begin{aligned} &Cycle : (z_k, \sigma_k^{\mathtt{T}}, z_{k+1}, \ldots\ldots z_l, \sigma_l^{\mathtt{T}}, z_k) \text{ where } z_i = (x_i, \overline{y_i}, y_i^1, y_i^2) \in \mathrm{Z} \text{ and} \\ &\sigma_i^{\mathtt{T}} = (\sigma_i, \sigma_i^1, \sigma_i^2) \in \Sigma^{\mathtt{T}}(i = k, k+1, \ldots., l). \end{aligned} \tag{4.8}$$

and an indeterminate cycle is given by a cycle such that

$$\exists i \in [k, l] \text{ such that } [(\overline{y_i} = F) \wedge (\sigma_i \neq \epsilon)] \tag{4.9}$$

If there is an indeterminate cycle in the testing automaton, then it holds that K is not codiagnosable for G and M_i.

4.3 Implementation in libFAUDES

In this section, the implementation of the algorithm given in [13] as a part of the libFAUDES software library is described. LibFAUDES is an open-source C++ software library developed for implementation of algorithms of discrete event systems. This discrete event systems library includes many C++ class and functions that implement general purpose operations and algorithms contributed by many researches. The algorithm in [13] is implemented by using these classes and functions. The algorithm consists of three steps as described in section-4.2.

4.3.1 Augmented Specification Automaton

The first step is constructing the augmented specification automaton $\overline{H}$ from the given specification automaton H. In libFAUDES, the function

```
Automaton(Generator &Gen)
```

can be directly used to construct $\overline{H}$. This function converts a generator into a formal automaton. That is, a new state,*dump* is added to the automaton and undefined transitions lead to the dump state.

4.3.2 Testing Automaton

The second step is constructing the testing automaton as explained in section-4.2.2. Equation-(4.6) and (4.7) include fundamental properties for this construction. These two equations are formulated in a very compact form. However, there are twenty-three cases derived from these two equations. In this thesis work, all conditions are examined and the following function is implemented in libFAUDES.

```
void TestingAutomaton (const System& Gen, const Generator& Spec,
const EventSet& ObsEvent1, const EventSet& ObsEvent2,
Generator& Testing)
```

This function generates the testing automaton using the plant automaton *Gen*, the specification automaton *Spec* and the local observation masks given by "ObsEvent1" and "ObsEvent2". As explained before, there are twenty-three cases to construct the testing automaton. All of these cases are examined below. The general working principle of the algorithm is outlined in algorithm-1.

```
Algorithm 1
Input: $G$, $K$, $M_1$, $M_2$
Initialization:
    1. Insert state $(x_0,\overline{y}_0,y_0,y_0)$ into state set $Z$ of $T$
    2. Insert state $(x_0,\overline{y}_0,y_0,y_0)$ into stack of waiting states waitingTesting.
    3. Empty set of already processed states doneTesting
Iterations:
while (waitingTesting $\neq \emptyset$)
    1. currentstate=waitingTesting.Top();
    2. doneTesting.Insert(currentstate);
    3. waitingTesting.Pop(currentstate);
    4. $x$=currentstate[0];  $\overline{y}$=currentstate[1];  $y^1$=currentstate[2];
       $y^2$=currentstate[3];
    5. Determine new states, events and transitions starting from currentstate =
       $(x,\overline{y},y^1,y^2)$. The different cases of this step are explained below
    6. Insert any new state that was found in step 5. into the state set $Z$ of T
    7. Insert any new state to waitingTesting if it does not exist in waitingTest-
       ing or doneTesting
    8. Insert any new event that was found in step 5. into the event set $\Sigma^T$ of T
    9. Insert any new transition that was found in step 5. into T
Return: Testing Automaton $T$
```

The details of the twenty-three cases for step 5. of Algorithm 1 are given below. Moreover, some cases are illustrated in figures. These figures are parts of the testing automaton construction from G in figure-3.5, K in figure-3.6 and the local observation masks in table-4.1.

1. If there is a transition from x with σ event in G and σ event is observable

Table4.1: Local Observation Masks

$\sigma \in \Sigma$	a	b	c	d	e	f	g
$M_1(\sigma)$	ϵ	ϵ	ϵ	A_1	B	ϵ	D_1
$M_2(\sigma)$	ϵ	ϵ	ϵ	A_2	ϵ	C	D_2

with M_1 and M_2 and there are transitions from y^1 with σ and from y^2 with σ in K, then

* $\{(x, \overline{y}, y^1, y^2),(\sigma, \sigma, \sigma),(x_{next}, \overline{y}_{next}, y^1_{next}, y^2_{next})\}$ transition, (σ, σ, σ) event and $(x_{next}, \overline{y}_{next}, y^1_{next}, y^2_{next})$ state are inserted to T. Moreover, $(x_{next}, \overline{y}_{next}, y^1_{next}, y^2_{next})$ state is pushed to *waitingTesting* queue if this state was not processed previously.

2. If there is a transition from x with σ event in G and σ event is observable with M_1 and M_2 and there are transitions from y^1 with σ, and from y^2 with σ_2 in K where $M_2(\sigma_2) = \epsilon$, then

   * $\{(x, \overline{y}, y^1, y^2), (\epsilon, \epsilon, \sigma_2), (x, \overline{y}, y^1, y^2_{next})\}$ transition, $(\epsilon, \epsilon, \sigma_2)$ event and $(x, \overline{y}, y^1, y^2_{next})$ state are inserted to T. Moreover, $(x, \overline{y}, y^1, y^2_{next})$ state is pushed to *waitingTesting* queue if this state was not processed previously.

3. If there is a transition from x with σ event in G and σ event is observable with M_1 and M_2 and there are transitions from y^1 with σ_1 and from y^2 with σ in K where $M_1(\sigma_1) = \epsilon$, then

   * $\{(x, \overline{y}, y^1, y^2), (\epsilon, \sigma_1, \epsilon), (x, \overline{y}, y^1_{next}, y^2)\}$ transition, $(\epsilon, \sigma_1, \epsilon)$ event and $(x, \overline{y}, y^1_{next}, y^2)$ state are inserted to T. Moreover, $(x, \overline{y}, y^1_{next}, y^2)$ state is pushed to *waitingTesting* queue if this state was not processed previously.

4. If there is a transition from x with σ event in G and σ event is observable with M_1 and M_2 and there are transitions from y^1 with σ_1 and from y^2 with σ_2 in K where $M_1(\sigma_1) = \epsilon$ and $M_2(\sigma_2) = \epsilon$, then

   * $\{(x, \overline{y}, y^1, y^2), (\epsilon, \sigma_1, \sigma_2), (x, \overline{y}, y^1_{next}, y^2_{next})\}$ transition, $(\epsilon, \sigma_1, \sigma_2)$ event and $(x, \overline{y}, y^1_{next}, y^2_{next})$ state are inserted to T.

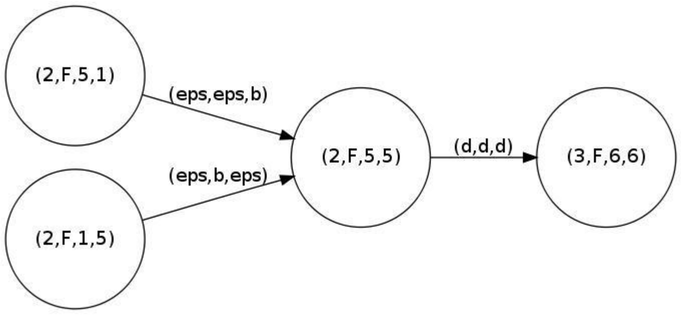

Figure 4.4: Testing automaton construction, cases 1,2,3

Moreover, $(x, \overline{y}, y^1_{next}, y^2_{next})$ state is pushed to *waitingTesting* queue if this state was not processed previously.

* $\{(x, \overline{y}, y^1, y^2), (\epsilon, \sigma_1, \epsilon), (x, \overline{y}, y^1_{next}, y^2)\}$ transition, $(\epsilon, \sigma_1, \epsilon)$ event and $(x, \overline{y}, y^1_{next}, y^2)$ state are inserted to T. Moreover, $(x, \overline{y}, y^1_{next}, y^2)$ state is pushed to *waitingTesting* queue if this state was not processed previously.

* $\{(x, \overline{y}, y^1, y^2), (\epsilon, \epsilon, \sigma_2), (x, \overline{y}, y^1, y^2_{next})\}$ transition, $(\epsilon, \epsilon, \sigma_2)$ event and $(x, \overline{y}, y^1, y^2_{next})$ state are inserted to T. Moreover, $(x, \overline{y}, y^1, y^2_{next})$ state is pushed to *waitingTesting* queue if this state was not processed previously.

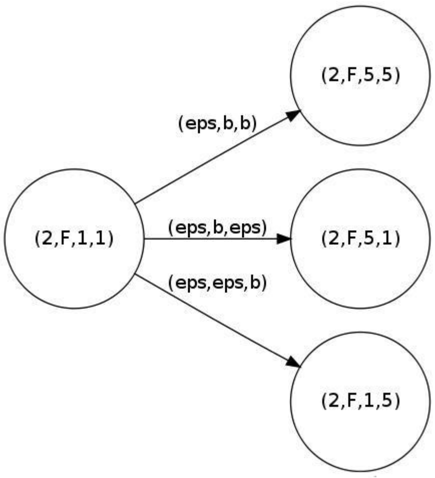

Figure 4.5: Testing automaton construction, case 4

5. If there is a transition from x with σ event in G and σ event is observable

with M_1 and M_2 and there is a transition from y^1 with σ_1 in K where $M_1(\sigma_1) = \epsilon$, and there is not any transition from y^2 in K, then

  * $\{(x, \overline{y}, y^1, y^2), (\epsilon, \sigma_1, \epsilon), (x, \overline{y}, y^1_{next}, y^2)\}$ transition, $(\epsilon, \sigma_1, \epsilon)$ event and $(x, \overline{y}, y^1_{next}, y^2)$ state are inserted to T. Moreover, $(x, \overline{y}, y^1_{next}, y^2)$ state is pushed to *waitingTesting* queue if this state was not processed previously.

6. If there is a transition from x with σ event in G and σ event is observable with M_1 and M_2 and there is a transition from y^2 with σ_2 in K where $M_2(\sigma_2) = \epsilon$, and there is not any transition from y^1 in K, then

  * $\{(x, \overline{y}, y^1, y^2), (\epsilon, \epsilon, \sigma_2), (x, \overline{y}, y^1, y^2_{next})\}$ transition, $(\epsilon, \epsilon, \sigma_2)$ event and $(x, \overline{y}, y^1, y^2_{next})$ state are inserted to T. Moreover, $(x, \overline{y}, y^1, y^2_{next})$ state is pushed to *waitingTesting* queue if this state was not processed previously.

7. If there is a transition from x with σ event in G and σ event is observable with only M_1 and there are transitions from y^1 with σ, and from y^2 with σ_2 in K where $M_2(\sigma_2) = \epsilon$, then

  * $\{(x, \overline{y}, y^1, y^2), (\epsilon, \epsilon, \sigma_2), (x, \overline{y}, y^1, y^2_{next})\}$ transition, $(\epsilon, \epsilon, \sigma_2)$ event and $(x, \overline{y}, y^1, y^2_{next})$ state are inserted to T. Moreover, $(x, \overline{y}, y^1, y^2_{next})$ state is pushed to *waitingTesting* queue if this state was not processed previously.
  * $\{(x, \overline{y}, y^1, y^2), (\sigma, \sigma, \sigma_2), (x_{next}, \overline{y}_{next}, y^1_{next}, y^2_{next})\}$ transition, $(\sigma, \sigma, \sigma_2)$ event and $(x_{next}, \overline{y}_{next}, y^1_{next}, y^2_{next})$ state are inserted to T. Moreover, $(x_{next}, \overline{y}_{next}, y^1_{next}, y^2_{next})$ state is pushed to *waitingTesting* queue if this state was not processed previously.
  * $\{(x, \overline{y}, y^1, y^2), (\sigma, \sigma, \epsilon), (x_{next}, \overline{y}_{next}, y^1_{next}, y^2)\}$ transition, $(\sigma, \sigma, \epsilon)$ event and $(x_{next}, \overline{y}_{next}, y^1_{next}, y^2)$ state are inserted to T. Moreover, $(x_{next}, \overline{y}_{next}, y^1_{next}, y^2)$ state is pushed to *waitingTesting* queue if this state was not processed previously.

8. If there is a transition from x with σ event in G and σ event is observable with only M_1 and there is a transition from y^1 with σ in K, and there is

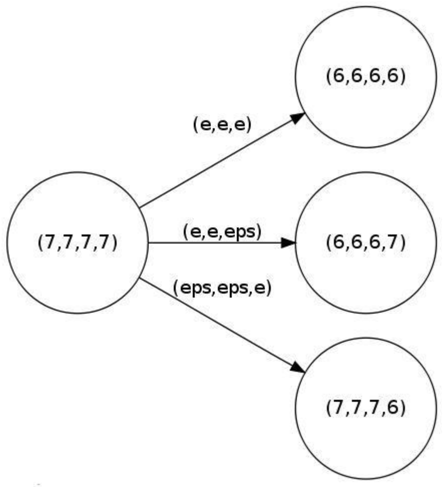

Figure 4.6: Testing automaton construction, case 7

not any transition from y^2 in K, then

- $\{(x, \overline{y}, y^1, y^2), (\sigma, \sigma, \epsilon), (x_{next}, \overline{y}_{next}, y^1_{next}, y^2)\}$ transition, $(\sigma, \sigma, \epsilon)$ event and $(x_{next}, \overline{y}_{next}, y^1_{next}, y^2)$ state are inserted to T. Moreover, $(x_{next}, \overline{y}_{next}, y^1_{next}, y^2)$ state is pushed to *waitingTesting* queue if this state was not processed previously.

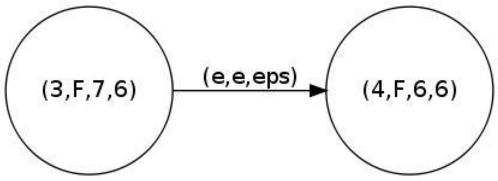

Figure 4.7: Testing automaton construction, case 8

9. If there is a transition from x with σ event in G and σ event is observable with only M_1 and there are transitions from y^1 with σ_1, and from y^2 with σ_2 in K where $M_1(\sigma_1) = \epsilon$ and $M_2(\sigma_2) = \epsilon$, then

   - $\{(x, \overline{y}, y^1, y^2), (\epsilon, \sigma_1, \sigma_2), (x, \overline{y}, y^1_{next}, y^2_{next})\}$ transition, $(\epsilon, \sigma_1, \sigma_2)$ event and $(x, \overline{y}, y^1_{next}, y^2_{next})$ state are inserted to T. Moreover, $(x, \overline{y}, y^1_{next}, y^2_{next})$ state is pushed to *waitingTesting* queue if this state was not processed previously.

* $\{(x, \overline{y}, y^1, y^2), (\epsilon, \epsilon, \sigma_2), (x, \overline{y}, y^1, y^2_{next})\}$ transition, $(\epsilon, \epsilon, \sigma_2)$ event and $(x, \overline{y}, y^1, y^2_{next})$ state are inserted to T. Moreover, $(x, \overline{y}, y^1, y^2_{next})$ state is pushed to *waitingTesting* queue if this state was not processed previously.

* $\{(x, \overline{y}, y^1, y^2), (\epsilon, \sigma_1, \epsilon), (x, \overline{y}, y^1_{next}, y^2)\}$ transition, $(\epsilon, \sigma_1, \epsilon)$ event and $(x, \overline{y}, y^1_{next}, y^2)$ state are inserted to T. Moreover, $(x, \overline{y}, y^1_{next}, y^2)$ state is pushed to *waitingTesting* queue if this state was not processed previously.

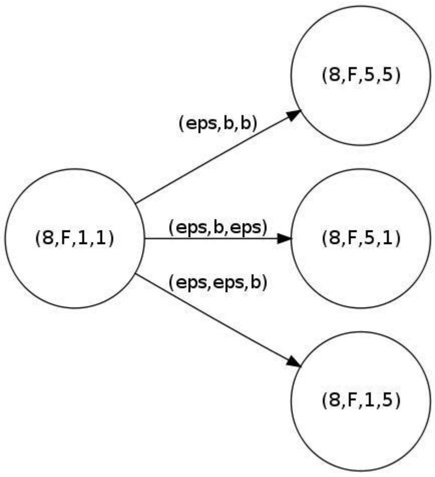

Figure 4.8: Testing automaton construction, case 9

10. If there is a transition from x with σ event in G and σ event is observable with only M_1 and there is a transition from y^1 with σ_1 in K where $M_1(\sigma_1) = \epsilon$, and there is not any transition from y^2 in K, then

    * $\{(x, \overline{y}, y^1, y^2), (\epsilon, \sigma_1, \epsilon), (x, \overline{y}, y^1_{next}, y^2)\}$ transition, $(\epsilon, \sigma_1, \epsilon)$ event and $(x, \overline{y}, y^1_{next}, y^2)$ state are inserted to T. Moreover, $(x, \overline{y}, y^1_{next}, y^2)$ state is pushed to *waitingTesting* queue if this state was not processed previously.

11. If there is a transition from x with σ event in G and σ event is observable with only M_1 and there is a transition from y^2 with σ_2 in K where $M_2(\sigma_2) = \epsilon$ and there is not any transition from y^1 in K, then

    * $\{(x, \overline{y}, y^1, y^2), (\epsilon, \epsilon, \sigma_2), (x, \overline{y}, y^1, y^2_{next})\}$ transition,

$(\epsilon, \epsilon, \sigma_2)$ event and $(x, \overline{y}, y^1, y^2_{next})$ state are inserted to T. Moreover, $(x, \overline{y}, y^1, y^2_{next})$ state is pushed to *waitingTesting* queue if this state was not processed previously.

12. If there is a transition from x with σ event in G and σ event is observable with only M_2 and there are transitions from y^2 with σ, and from y^1 with σ_1 in K where $M_1(\sigma_1) = \epsilon$, then

    * $\{(x, \overline{y}, y^1, y^2), (\sigma, \sigma_1, \sigma), (x_{next}, \overline{y}_{next}, y^1_{next}, y^2_{next})\}$ transition, $(\sigma, \sigma_1, \sigma)$ event and $(x_{next}, \overline{y}_{next}, y^1_{next}, y^2_{next})$ state are inserted to T. Moreover, $(x_{next}, \overline{y}_{next}, y^1_{next}, y^2_{next})$ state is pushed to *waitingTesting* queue if this state was not processed previously.
    * $\{(x, \overline{y}, y^1, y^2), (\epsilon, \sigma_1, \epsilon), (x, \overline{y}, y^1_{next}, y^2)\}$ transition, $(\epsilon, \sigma_1, \epsilon)$ event and $(x, \overline{y}, y^1_{next}, y^2)$ state are inserted to T. Moreover, $(x, \overline{y}, y^1_{next}, y^2)$ state is pushed to *waitingTesting* queue if this state was not processed previously.
    * $\{(x, \overline{y}, y^1, y^2), (\sigma, \epsilon, \sigma), (x_{next}, \overline{y}_{next}, y^1, y^2_{next})\}$ transition, $(\sigma, \epsilon, \sigma)$ event and $(x_{next}, \overline{y}_{next}, y^1, y^2_{next})$ state are inserted to T. Moreover, $(x_{next}, \overline{y}_{next}, y^1, y^2_{next})$ state is pushed to *waitingTesting* queue if this state was not processed previously.

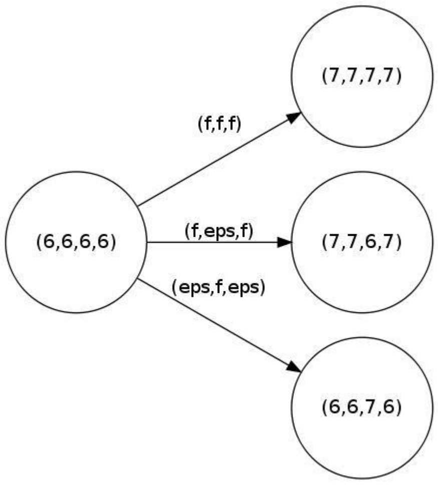

Figure 4.9: Testing automaton construction, case 12

13. If there is a transition from x with σ event in G and σ event is observable

with only M_2 and there is a transition from y^1 with σ in K, and there is not any transition from y^2 in K, then

- $\{(x, \overline{y}, y^1, y^2), (\sigma, \epsilon, \sigma), (x_{next}, \overline{y}_{next}, y^1, y^2_{next})\}$ transition, $(\sigma, \epsilon, \sigma)$ event and $(x_{next}, \overline{y}_{next}, y^1, y^2_{next})$ state are inserted to T. Moreover, $(x_{next}, \overline{y}_{next}, y^1, y^2_{next})$ state is pushed to *waitingTesting* queue if this state was not processed previously.

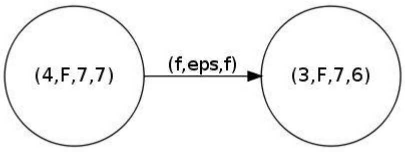

Figure 4.10: Testing automaton construction, case 13

14. If there is a transition from x with σ event in G and σ event is observable with only M_2 and there are transitions from y^2 with σ_2, and from y^1 with σ_1 in K where $M_1(\sigma_1) = \epsilon$ and $M_2(\sigma_2) = \epsilon$, then

    - $\{(x, \overline{y}, y^1, y^2), (\epsilon, \epsilon, \sigma_2), (x, \overline{y}, y^1, y^2_{next})\}$ transition, $(\epsilon, \epsilon, \sigma_2)$ event and $(x, \overline{y}, y^1, y^2_{next})$ state are inserted to T. Moreover, $(x, \overline{y}, y^1, y^2_{next})$ state is pushed to *waitingTesting* queue if this state was not processed previously.
    - $\{(x, \overline{y}, y^1, y^2), (\epsilon, \sigma_1, \epsilon), (x, \overline{y}, y^1_{next}, y^2)\}$ transition, $(\epsilon, \sigma_1, \epsilon)$ event and $(x, \overline{y}, y^1_{next}, y^2)$ state are inserted to T. Moreover, $(x, \overline{y}, y^1_{next}, y^2)$ state is pushed to *waitingTesting* queue if this state was not processed previously.
    - $\{(x, \overline{y}, y^1, y^2), (\epsilon, \sigma_1, \sigma_2), (x, \overline{y}, y^1_{next}, y^2_{next})\}$ transition, $(\epsilon, \sigma_1, \sigma_2)$ event and $(x, \overline{y}, y^1_{next}, y^2_{next})$ state are inserted to T. Moreover, $(x, \overline{y}, y^1_{next}, y^2_{next})$ state is pushed to *waitingTesting* queue if this state was not processed previously.

15. If there is a transition from x with σ event in Generator σ event is observable with only M_2 and there is not any transition from y^2 in K, and there is a transition from y^1 with σ_1 in K where $M_1(\sigma_1) = \epsilon$, then

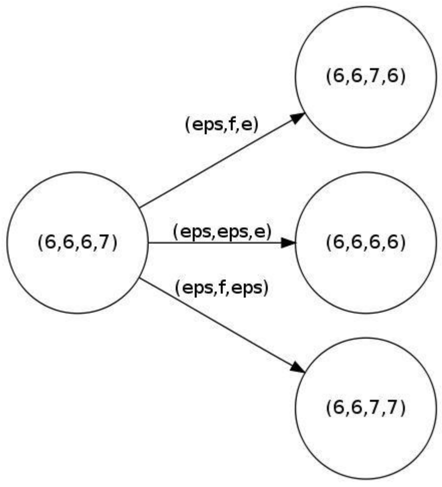

Figure 4.11: Testing automaton construction, case 14

- $\{(x, \overline{y}, y^1, y^2), (\epsilon, \sigma_1, \epsilon), (x, \overline{y}, y^1_{next}, y^2)\}$ transition, $(\epsilon, \sigma_1, \epsilon)$ event and $(x, \overline{y}, y^1_{next}, y^2)$ state are inserted to T. Moreover, $(x, \overline{y}, y^1_{next}, y^2)$ state is pushed to *waitingTesting* queue if this state was not processed previously.

16. If there is a transition from x with σ event in G and σ event is observable with only M_2 and there is not any transition from y^1 in K, and there is a transition from y^2 with σ_2 in K where $M_2(\sigma_2) = \epsilon$, then

    - $\{(x, \overline{y}, y^1, y^2), (\epsilon, \epsilon, \sigma_2), (x, \overline{y}, y^1, y^2_{next})\}$ transition, $(\epsilon, \epsilon, \sigma_2)$ event and $(x, \overline{y}, y^1, y^2_{next})$ state are inserted to T. Moreover, $(x, \overline{y}, y^1, y^2_{next})$ state is pushed to *waitingTesting* queue if this state was not processed previously.

17. If there is a transition from x with σ event in G and σ is not observable with neither M_1 nor M_2 and there is transition from y^1 with σ_1 in K where $M_1(\sigma_1) = \epsilon$, and there is not any transition from y^2 in K, then

    - $\{(x, \overline{y}, y^1, y^2), (\sigma, \epsilon, \epsilon), (x_{next}, \overline{y}_{next}, y^1, y^2)\}$ transition, $(\sigma, \epsilon, \epsilon)$ event and $(x_{next}, \overline{y}_{next}, y^1, y^2)$ state are inserted to T. Moreover, $(x_{next}, \overline{y}_{next}, y^1, y^2)$ state is pushed to *waitingTesting* queue if this state was not processed previously.
    - $\{(x, \overline{y}, y^1, y^2), (\epsilon, \sigma_1, \epsilon), (x, \overline{y}, y^1_{next}, y^2)\}$ transition,

$(\epsilon, \sigma_1, \epsilon)$ event and $(x, \overline{y}, y^1_{next}, y^2)$ state are inserted to T. Moreover, $(x, \overline{y}, y^1_{next}, y^2)$ state is pushed to *waitingTesting* queue if this state was not processed previously.

* $\{(x, \overline{y}, y^1, y^2), (\sigma, \sigma_1, \epsilon), (x_{next}, \overline{y}_{next}, y^1_{next}, y^2)\}$ transition, $(\sigma, \sigma_1, \epsilon)$ event and $(x_{next}, \overline{y}_{next}, y^1_{next}, y^2)$ state are inserted to T. Moreover, $(x_{next}, \overline{y}_{next}, y^1_{next}, y^2)$ state is pushed to *waitingTesting* queue if this state was not processed previously.

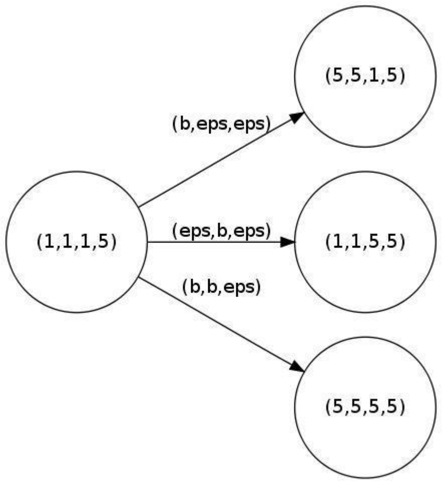

Figure 4.12: Testing automaton construction, case 17

18. If there is a transition from x with σ event in G and σ is not observable with neither M_1 nor M_2 and there is not any transition from y^1 in K and there is a transition from y^2 with σ_2 in K where $M_2(\sigma_2) = \epsilon$, then

* $\{(x, \overline{y}, y^1, y^2), (\sigma, \epsilon, \epsilon), (x_{next}, \overline{y}_{next}, y^1, y^2)\}$ transition, $(\sigma, \epsilon, \epsilon)$ event and $(x_{next}, \overline{y}_{next}, y^1, y^2)$ state are inserted to T. Moreover, $(x_{next}, \overline{y}_{next}, y^1, y^2)$ state is pushed to *waitingTesting* queue if this state was not processed previously.
* $\{(x, \overline{y}, y^1, y^2), (\epsilon, \epsilon, \sigma_2), (x, \overline{y}, y^1, y^2_{next})\}$ transition, $(\epsilon, \epsilon, \sigma_2)$ event and $(x, \overline{y}, y^1, y^2_{next})$ state are inserted to T. Moreover, $(x, \overline{y}, y^1, y^2_{next})$ state is pushed to *waitingTesting* queue if this state was not processed previously.
* $\{(x, \overline{y}, y^1, y^2), (\sigma, \epsilon, \sigma_2), (x_{next}, \overline{y}_{next}, y^1, y^2_{next})\}$ transition,

$(\sigma, \epsilon, \sigma_2)$ event and $(x_{next}, \overline{y}_{next}, y^1, y^2_{next})$ state are inserted to T. Moreover, $(x_{next}, \overline{y}_{next}, y^1, y^2_{next})$ state is pushed to *waitingTesting* queue if this state was not processed previously.

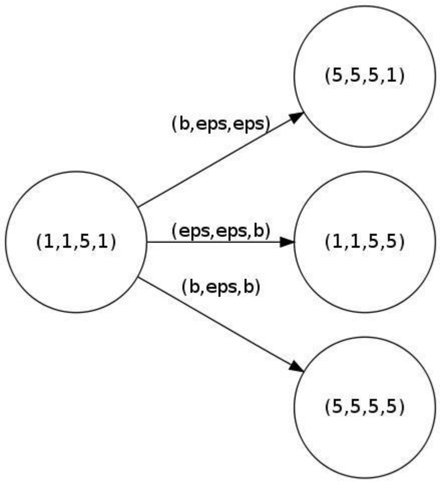

Figure 4.13: Testing automaton construction, case 18

19. If there is a transition from x with σ event in G and σ is not observable with neither M_1 nor M_2 and There is not any function transition from y^1 and from y^2 in K, then

   * $\{(x, \overline{y}, y^1, y^2), (\sigma, \epsilon, \epsilon), (x_{next}, \overline{y}_{next}, y^1, y^2)\}$ transition, $(\sigma, \epsilon, \epsilon)$ event and $(x_{next}, \overline{y}_{next}, y^1, y^2)$ state are inserted to T. Moreover, $(x_{next}, \overline{y}_{next}, y^1, y^2)$ state is pushed to *waitingTesting* queue if this state was not processed previously.

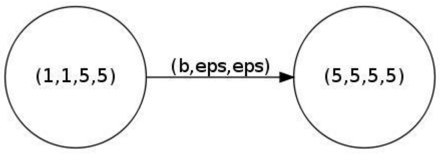

Figure 4.14: Testing automaton construction, case 19

20. If there is not any transition from x with σ event in G and there are transitions from y^1 with σ_1 and from y^2 with σ_2 in K where $M_1(\sigma_1) = \epsilon$ and $M_2(\sigma_2) = \epsilon$, then

* $\{(x, \overline{y}, y^1, y^2), (\epsilon, \sigma_1, \epsilon), (x, \overline{y}, y^1_{next}, y^2)\}$ transition,
  $(\epsilon, \sigma_1, \epsilon)$ event and $(x, \overline{y}, y^1_{next}, y^2)$ state are inserted to T.
  Moreover, $(x, \overline{y}, y^1_{next}, y^2)$ state is pushed to *waitingTesting* queue if this state was not processed previously.
* $\{(x, \overline{y}, y^1, y^2), (\epsilon, \epsilon, \sigma_2), (x, \overline{y}, y^1, y^2_{next})\}$ transition,
  $(\epsilon, \epsilon, \sigma_2)$ event and $(x, \overline{y}, y^1, y^2_{next})$ state are inserted to T.
  Moreover, $(x, \overline{y}, y^1, y^2_{next})$ state is pushed to *waitingTesting* queue if this state was not processed previously.
* $\{(x, \overline{y}, y^1, y^2), (\epsilon, \sigma_1, \sigma_2), (x, \overline{y}, y^1_{next}, y^2_{next})\}$ transition,
  $(\epsilon, \sigma_1, \sigma_2)$ event and $(x, \overline{y}, y^1_{next}, y^2_{next})$ state are inserted to T.
  Moreover, $(x, \overline{y}, y^1_{next}, y^2_{next})$ state is pushed to *waitingTesting* queue if this state was not processed previously.

21. If there is not any transition from x with σ event in G and there is a transition from y^1 with σ_1 in K where $M_1(\sigma_1) = \epsilon$, and there is not any transition from y^2 in K, then

    * $\{(x, \overline{y}, y^1, y^2), (\epsilon, \sigma_1, \epsilon), (x, \overline{y}, y^1_{next}, y^2)\}$ transition,
      $(\epsilon, \sigma_1, \epsilon)$ event and $(x, \overline{y}, y^1_{next}, y^2)$ state are inserted to T.
      Moreover, $(x, \overline{y}, y^1_{next}, y^2)$ state is pushed to *waitingTesting* queue if this state was not processed previously.

22. If there is not any transition from x with σ event in G and there is not any transition from y^1 in K and there is a transition from y^2 with σ_2 in K where $M_2(\sigma_2) = \epsilon$, then

    * $\{(x, \overline{y}, y^1, y^2), (\epsilon, \epsilon, \sigma_2), (x, \overline{y}, y^1, y^2_{next})\}$ transition,
      $(\epsilon, \epsilon, \sigma_2)$ event and $(x, \overline{y}, y^1, y^2_{next})$ state are inserted to T.
      Moreover, $(x, \overline{y}, y^1, y^2_{next})$ state is pushed to *waitingTesting* queue if this state was not processed previously.

23. If there is a transition from x with σ event in G and σ is not observable with neither M_1 nor M_2 and there are transitions from y^1 with σ_1 and from y^2 with σ_2 in K where $M_1(\sigma_1) = \epsilon$ and $M_2(\sigma_2) = \epsilon$, then

    * $\{(x, \overline{y}, y^1, y^2), (\sigma, \epsilon, \epsilon), (x_{next}, \overline{y}_{next}, y^1, y^2)\}$ transition,
      $(\sigma, \epsilon, \epsilon)$ event and $(x_{next}, \overline{y}_{next}, y^1, y^2)$ state are inserted to T.

Moreover, $(x_{next}, \overline{y}_{next}, y^1, y^2)$ state is pushed to *waitingTesting* queue if this state was not processed previously.

- $\{(x, \overline{y}, y^1, y^2), (\epsilon, \sigma_1, \epsilon), (x, \overline{y}, y^1_{next}, y^2)\}$ transition,
  $(\epsilon, \sigma_1, \epsilon)$ event and $(x, \overline{y}, y^1_{next}, y^2)$ state are inserted to T.
  Moreover, $(x, \overline{y}, y^1_{next}, y^2)$ state is pushed to *waitingTesting* queue if this state was not processed previously.
- $\{(x, \overline{y}, y^1, y^2), (\epsilon, \epsilon, \sigma_2), (x, \overline{y}, y^1, y^2_{next})\}$ transition,
  $(\epsilon, \epsilon, \sigma_2)$ event and $(x, \overline{y}, y^1, y^2_{next})$ state are inserted to T.
  Moreover, $(x, \overline{y}, y^1, y^2_{next})$ state is pushed to *waitingTesting* queue if this state was not processed previously.
- $\{(x, \overline{y}, y^1, y^2), (\sigma, \sigma_1, \epsilon), (x_{next}, \overline{y}_{next}, y^1_{next}, y^2)\}$ transition,
  $(\sigma, \sigma_1, \epsilon)$ event and $(x_{next}, \overline{y}_{next}, y^1_{next}, y^2)$ state are inserted to T.
  Moreover, $(x_{next}, \overline{y}_{next}, y^1_{next}, y^2)$ state is pushed to *waitingTesting* queue if this state was not processed previously.
- $\{(x, \overline{y}, y^1, y^2), (\sigma, \epsilon, \sigma_2), (x_{next}, \overline{y}_{next}, y^1, y^2_{next})\}$ transition,
  $(\sigma, \epsilon, \sigma_2)$ event and $(x_{next}, \overline{y}_{next}, y^1, y^2_{next})$ state are inserted to T.
  Moreover, $(x_{next}, \overline{y}_{next}, y^1, y^2_{next})$ state is pushed to *waitingTesting* queue if this state was not processed previously.
- $\{(x, \overline{y}, y^1, y^2), (\epsilon, \sigma_1, \sigma_2), (x, \overline{y}, y^1_{next}, y^2_{next})\}$ transition,
  $(\epsilon, \sigma_1, \sigma_2)$ event and $(x, \overline{y}, y^1_{next}, y^2_{next})$ state are inserted to T.
  Moreover, $(x, \overline{y}, y^1_{next}, y^2_{next})$ state is pushed to *waitingTesting* queue if this state was not processed previously.
- $\{(x, \overline{y}, y^1, y^2), (\sigma, \sigma_1, \sigma_2), (x_{next}, \overline{y}_{next}, y^1_{next}, y^2_{next})\}$ transition,
  $(\sigma, \sigma_1, \sigma_2)$ event and $(x_{next}, \overline{y}_{next}, y^1_{next}, y^2_{next})$ state are inserted to T.
  Moreover, $(x_{next}, \overline{y}_{next}, y^1_{next}, y^2_{next})$ state is pushed to *waitingTesting* queue if this state was not processed previously.

In order to illustrate the testing automaton construction, the result T of applying the function `TestingAutomaton` to the plant G in figure-3.5, the specification automaton H in figure-3.6 and the observation masks in table-4.1 is shown in figure-4.16.

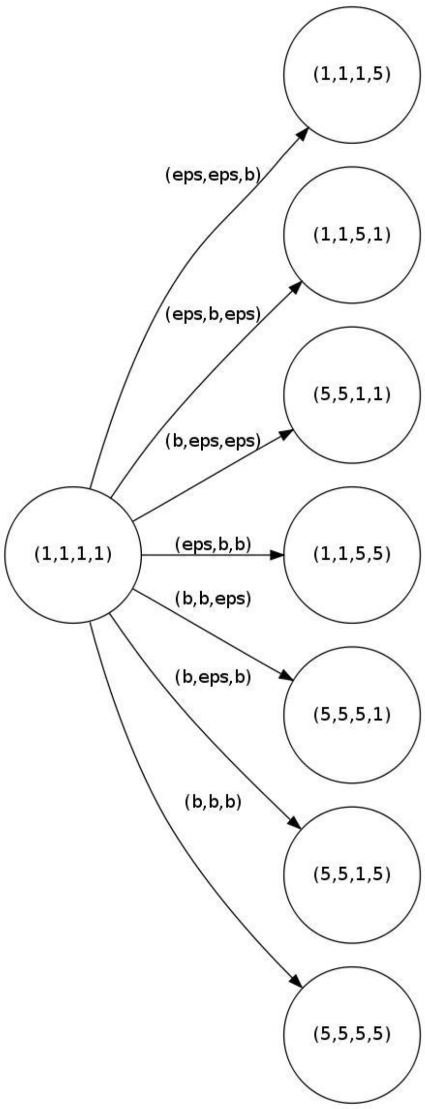

Figure 4.15: Testing automaton construction, case 23

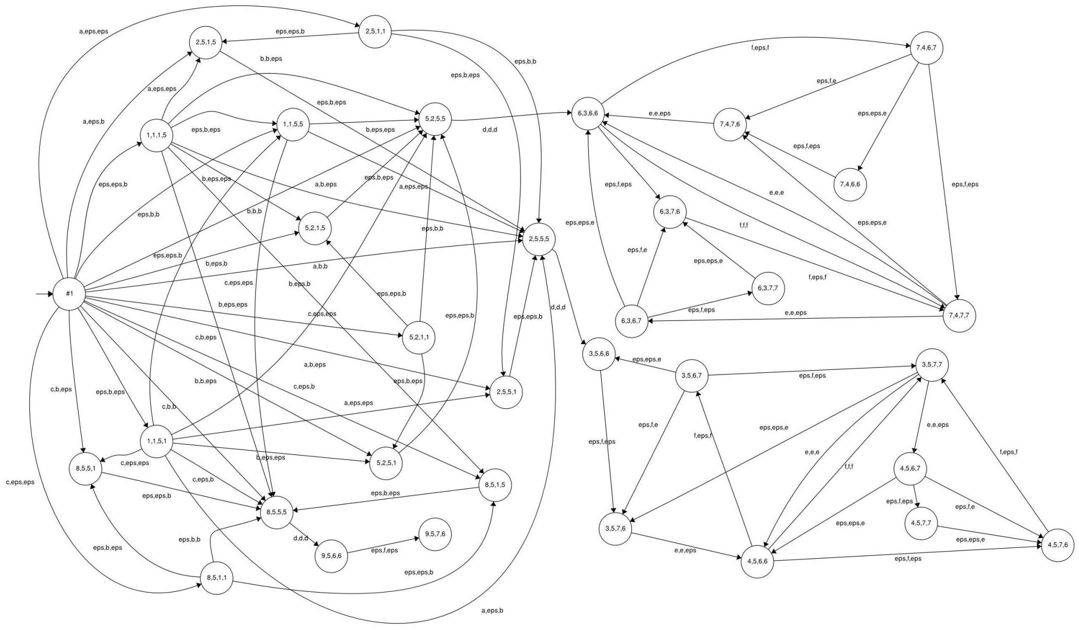

Figure 4.16: Testing Automaton

4.3.3 Checking the Violation of Codiagnosability

The third step to decide violation of co-diagnosability is searching for indeterminate cycles in the testing automaton. If such indeterminate cycles are found, co-diagnosability is violated. Hereby, the characteristic of an indeterminate cycle is that it has to contain at least one transitions with an event $\sigma^T = (\sigma, \sigma_1, \sigma_2) \in \Sigma^T$ such that $\sigma \neq \epsilon$ and that the second component of the testing automaton state in the cycle must be F. Therefore we should first remove all cycles from the testing automaton that do not fulfill these conditions. This is achieved by first Deleting all non-faulty states (second component is not f) in the testing automaton. This step can be achieved by using the function *bool DelState(Idx index)* in libFAUDES which deletes states from an automaton by index. We delete states whose second component is not equal to F.

After deleting non-faulty states, the resulting automaton only consists of faulty states. Next, we search transitions which events' first element is equal to ϵ. After finding these transitions, we construct new testing automaton by using equivalence relation described in section-5.2. Any cycle in this automaton is a candidate for an indeterminate cycle. By using the function *bool ComputeSCC(const Generator &rGen, const SccFilter &rFilter,std::list <StateSet> &rSccList, StateSet &rRoots)* in libFAUDES, all such cycles are found. The inputs of *ComputeSCC* are an automaton and filter type. We choose the testing automaton including only faulty states as the input automaton and "0x02" as input filter. When filter is set to 0x02, all cycles in the system are found and if there is any cycle in the automaton, *ComputeSCC* returns *true*. By using this function, we detect all loops in the testing automaton consisting of only faulty states. Therefore, if this function returns *true*, then this means that there is an indeterminate cycle in the testing automaton and the language is not codiagnosable. *ComputeSCC* function returns *true* if we apply the testing automaton given in figure-4.16. Because there is an indeterminate cycle in the testing automaton. This loop can be seen in figure-5.9. In this figure, states in the indeterminate cycle are shown with double circle.

We implement the steps checking codiagnosability with different generator automata, specification automata and observation masks. In the first application, the language generated by G, shown in figure-4.17 is codiagnosable with respect to specification automaton K in figure-4.18 and observation masks given in table-4.2 where a and f are unobservable events. It can be seen that the testing automaton's part consisting of only faulty states as seen in figure-4.19 does not include any cycle, that is, there is not any indeterminate cycle in the testing automaton. In order to interpret this result, we see that the first observation mask can detect b and c events. If a string includes the event c, but does not include the event b, it is said that this string is faulty. The second observation mask can detect c and d events. If a string includes d, then it detects that this string is faulty. All faulty strings can be detected by local observation masks, therefore the language generated by G, is codiagnosable.

Table4.2: Local Observation Masks

$\sigma \in \Sigma$	a	b	c	d	f
$M_1(\sigma)$	ϵ	B	C	ϵ	ϵ
$M_2(\sigma)$	ϵ	ϵ	C	D	ϵ

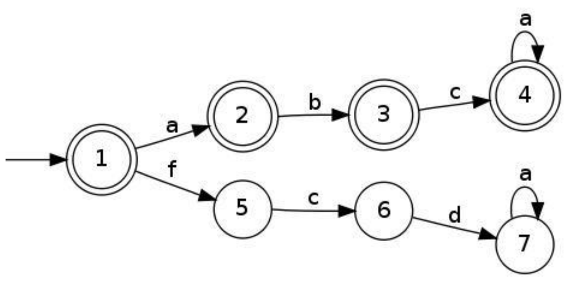

Figure 4.17: Codiagnosable Generator,G

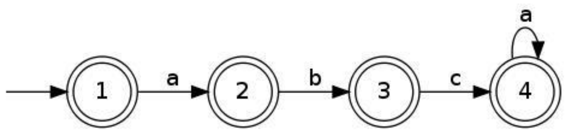

Figure 4.18: Specification,K

In the second example, the language generated by G, shown in figure-4.20 is not codiagnosable with respect to specification automaton K in figure-4.21 and observation masks function given in table-4.3 where s and f are unobservable events. This generator is not codiagnosable, because generated testing automaton includes an indeterminate cycle. This cycle is given in figure-4.22.

Table4.3: Local Observation Masks

$\sigma \in \Sigma$	a	b	c	d	e	f	s
$M_1(\sigma)$	A	B	ϵ	D	ϵ	ϵ	ϵ
$M_2(\sigma)$	A	ϵ	C	ϵ	E	ϵ	ϵ

In the last example, the plant automaton G includes deadlock states, state-*4* and state-*7*. If the system reaches these states, then it remains there. The testing automaton constructed from generator G given in figure-5.18, specification automaton K shown in figure-5.19 and observation masks given in table-5.2 where a and f are unobservable events and f state is faulty. In [13] Qui and Kumar suggest that, if there is any deadlock state, add self loop with ϵ event. After applying this trick testing automaton consisting of only F states is shown in figure-5.20. By looking this figure, it can be said that this system is codiagnosable, as there is not any cycle. However, it is clear that this generator is not

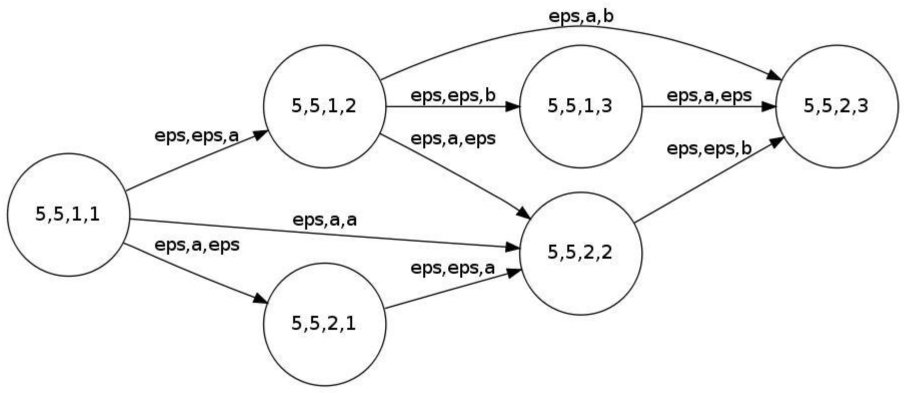

Figure 4.19: Testing Automaton with only Faulty States

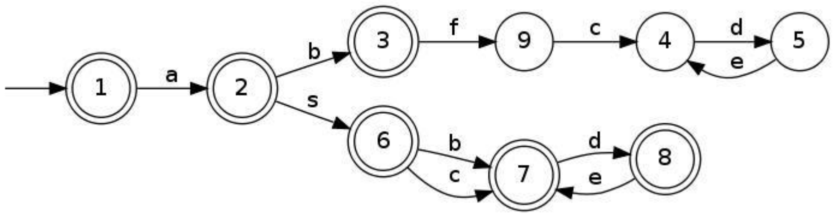

Figure 4.20: Non-codiagnosable plant G

codiagnosable. This contradiction shows that the trick suggested in [13] is not correct.

We propose a new trick for deadlocking states. We can add self-loop with new defined unobservable event, but not ϵ event, then we have deadlock-free generator and this generator does not give any wrong information about codiagnosability. The result of this operation is given in figure-4.23. If we apply this trick, testing automaton consisting of only failure states becomes like in figure-5.21. *ComputeSCC* function returns *true*, because this system is not codiagnosable.

4.4 Complexity Analysis

The complexity analysis of this algorithm is adapted from [13]. Let $|X|$ be the size of state set of generator and $|Y|$ be the size of state set of specification automaton and m be the number of local observation masks. The plant au-

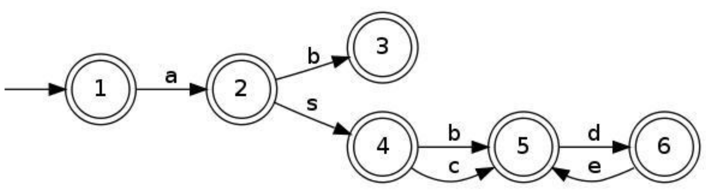

Figure 4.21: Specification automaton C

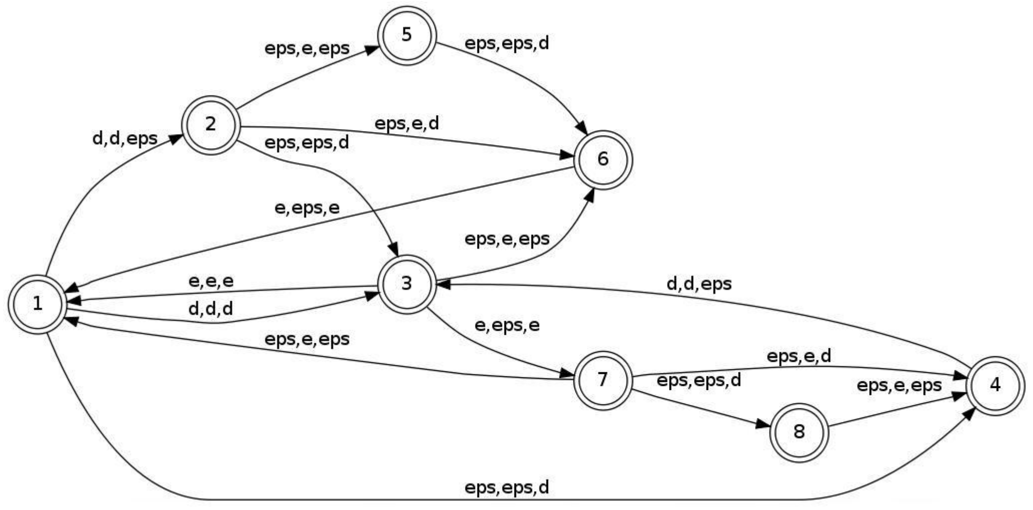

Figure 4.22: Indeterminate cycle in the testing automaton

tomaton can have at most |X|× |Σ| transitions and the specification automaton includes at most |Y| × |Σ| transitions. The augmented specification automaton and specification automaton have the same order of states and transitions.

The testing automaton states consist of one state from the plant automaton, one state from the augmented specification automaton and one state from the specification automaton for each local observation mask. Therefore, the testing automaton has at most $|X| \times |Y|^{\mathtt{m+1}}$ states and at each state of the testing automaton, there are at most $(|\Sigma + 1|)^{\mathtt{m+1}}$ transitions.

There are three steps to construct the testing automaton. The complexity of constructing the augmented specification automaton depends on the state and transition number of the specification automaton. Therefore the complexity of the first step is $O(|Y| \times |\Sigma|)$. The complexity of constructing the testing automaton and detecting indeterminate cycles in the testing automaton depends

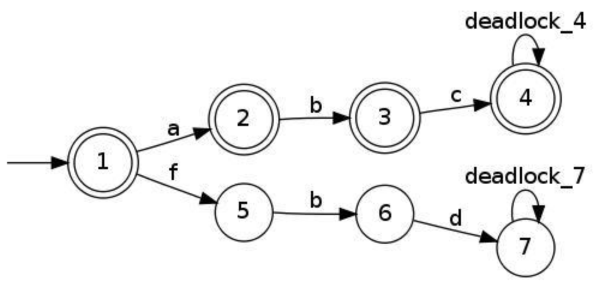

Figure 4.23: Generator which is deadlock-free

on the state and transition number of the testing automaton. Therefore the complexity of the second and third step is $O(|X| \times |Y|^{\mathtt{m+1}} \times (|\Sigma|+1)^{\mathtt{m+1}})$. Together, the complexity of the algorithm is equal to $O(|X| \times |Y|^{\mathtt{m+1}} \times (|\Sigma| + 1)^{\mathtt{m+1}})$.

CHAPTER 5

COMBINATION OF CENTRALIZED AND DECENTRALIZED DIAGNOSIS ALGORITHM

Centralized diagnosis and decentralized diagnosis are studied in the previous chapters. In centralized diagnosis algorithm faulty occurrences are detected by one observation mask in global system. In contrast, in decentralized diagnosis, fault occurrences have to be detected by more than one observation mask using decentralized observation. In this chapter we propose a new method that combines both diagnosis ideas.In particular, we suggest to use *decentralized diagnosis if possible and to use centralized diagnosis only if it is necessary.* To this end, we first establish the concept of a *supremal codiagnosable sublanguage* (SupCoDiag) of a system that contains all faulty strings that can be diagnosed using decentralized diagnosis. Then, we use centralized diagnosis only for faulty strings that do not belong to SupCoDiag. We further develop a polynomial-time algorithm for the computation of SupCoDiag.

5.1 Supremal Co-Diagnosable Sublanguage

We introduce the notion of a *co-diagnosable sublanguage* that contains all faulty strings whose occurrence can be detected by decentralized diagnosis as defined in [21]. As shown in [21], codiagnosable sublanguages of a plant automaton are closed under union. Therefore, let $L_1 \subseteq L$ be codiagnosable with respect to specification automaton, local observation masks and let L_2 be another sublanguage of plant language be codiagnosable with respect to specification automaton, local observation mask, we know that these sublanguages are closed under union, therefore $L_1 \cup L_2$ is codiagnosable with respect to specification automaton and local observation masks. We aim to find the largest sublanguage which is codiagnosable to use decentralized diagnoser the most. Therefore, the union of all codiagnosable sublanguages gives the supremal codiagnosable sublanguage. We note that the specification language is always a subset of the supremal codiagnosable sublanguage.

Let G be a plant automaton, $K \subseteq \Sigma^\star$ a specification and M_i *i=1,2* observation masks. Write

$$C = \{L' \subseteq L(G) | L' \text{ is a co-diagnosable sublanguage for } K \text{ and } M_i, \text{ i=1,2}\} \tag{5.1}$$

Then, $\mathit{SupCoD}(K, G, M_1, M_2) := \cup_{L' \in C} L'$ is the supremal element of C. We symbolized $\mathit{SupCoD}(K, G, M_1, M_2)$ as L^{co}. That is, a decentralized diagnoser can detect all faulty strings in $\mathit{SupCoD}(K, G, M_1, M_2) \setminus K$. On the other hand, the faulty strings in $L(G) \setminus \mathit{SupCoD}(K, G, M_1, M_2)$ cannot be detected by a decentralized diagnoser. This type of faults can in the best case be detected by a centralized diagnoser.

We identify the non-co-diagnosable sublanguage of *L(G)* as $L^{non} = K \cup (L(G) \setminus L^{co})$. That is any string in $L^{non} \setminus K$ represents the occurrences of failure that can not be detected by decentralized diagnosis.

From the previous analysis, we know that G given in figure-3.5 is not codiagnosable with respect to K given in figure-3.6 and local observation masks given in table-4.1. As described in section-4.3.3, there is an indeterminate cycle in the testing automaton for this example. In the literature, after verifying this system is not codiagnosable, we should diagnose the plant with centralized diagnoser. However, we suggest dividing the plant given in figure-3.5 into two subparts. These parts are given in figure-5.1 and in figure-5.2.

The testing automaton of the supremal codiagnosable sublanguge is given in figure-5.3. It can be seen that there is not any indeterminate cycle in the testing automaton. Therefore, this part is indeed codiagnosable. However, the testing

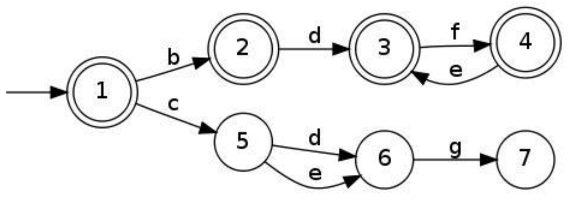

Figure 5.1: Supremal Codiagnosable Subpart of Plant

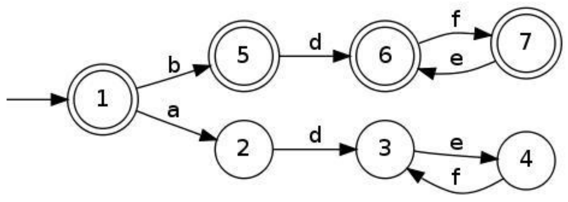

Figure 5.2: Nonco-diagnosable Subpart of Plant

automaton of the noncodiagnosable part is shown in figure-5.4. In this testing automaton, there is an indeterminate cycle which the states that are marked with double circles. Thus, we should attempt diagnosing this part of the plant using a centralized diagnoser.

5.2 Implementation in libFAUDES

This algorithm aims to use decentralized diagnosis where it is possible and to use centralized diagnosis where it is necessary. Therefore, we should find the minimal sublanguage of the plant that needs to be diagnosed by a centralized diagnoser, which is equivalent to finding the supremal codiagnosable sublanguage. More precisely, the part of the plant automaton that causes indeterminate cycles in the testing automaton needs to be identified. To achieve this goal, we first construct the testing automaton and detect all indeterminate cycles. If there is not any such indeterminate cycle, we conclude that codiagnosability is fulfilled. Otherwise, we should construct a new automaton that characterizes all plant strings that lead to indeterminate cycles. To this end, we point out several observations that will be used for the computation of the supremal codiagnosable sublanguage.

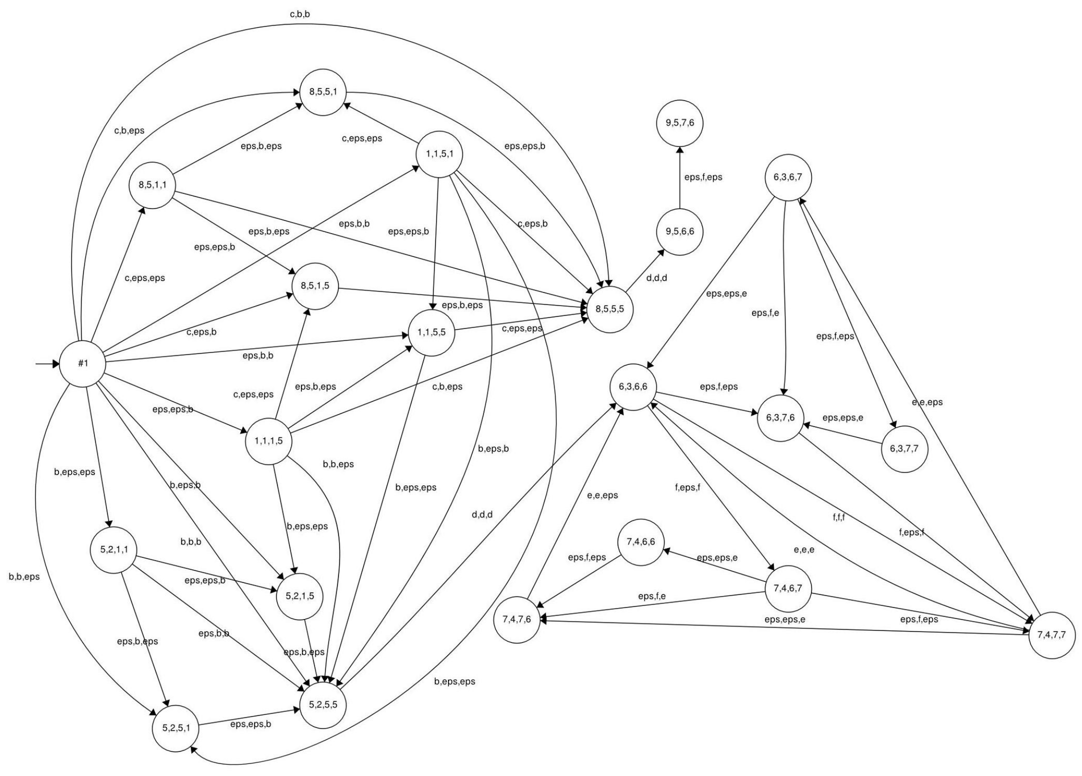

Figure 5.3: Testing Automaton of Supremal Codiagnosable Subpart

We consider the testing automaton T and introduce the set of faulty states $Z^F \subseteq Z$, the set of indeterminate cycle states $\mathcal{C}^T$ and the map $c_G : (\Sigma^T)^\star \to \Sigma^\star$ as follows:

$$Z^F = \{z \in Z | z = (-, F, -, -)\}$$
$$\mathcal{C}^T = \{z \in Z | z \text{ belongs to an indeterminate cycle}\}$$
$$\text{for all } s^T = (\sigma_1, -, -)(\sigma_2, -, -)\cdots(\sigma_m, -, -) \in (\Sigma^T)^\star, c_G(s^T) = \sigma_1\sigma_2\cdots\sigma_m.$$

That is, c_G projects each string in $(\Sigma^T)^\star$ to its first event component. Using this notation, the following statements about the testing automaton T can be deduced.

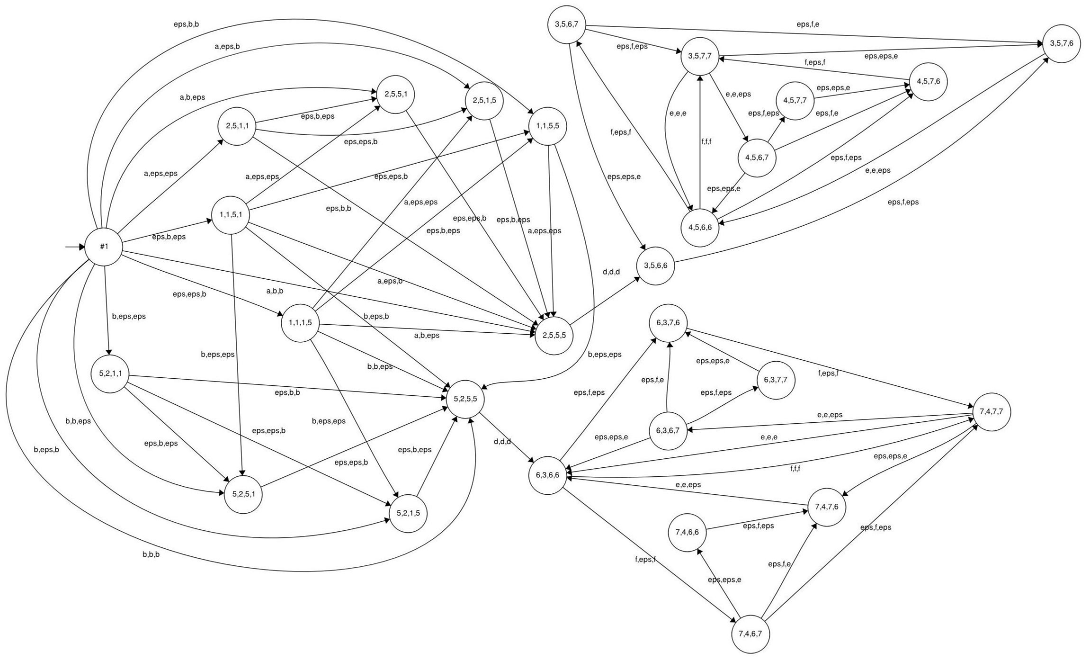

Figure 5.4: Testing Automaton of Noncodiagnosable Subpart

$$\forall s^T \in L(T) : c_G(s^{\mathtt{T}}) \in L(G) \tag{5.2}$$

$$\forall s^T \in L(T) : \gamma(z_0, s^{\mathtt{T}}) \in Z^F \Rightarrow c_G(s^{\mathtt{T}}) \notin K \tag{5.3}$$

$$\forall s^{\mathtt{T}} \in L(T) \quad \begin{array}{c}\text{such that } \gamma(z_0, s^{\mathtt{T}}) \in \mathcal{C}^T, \text{ it holds that } c_G(\hat{s}^T) \text{ is not} \\ \text{codiagnosable for all } \hat{s}^T \leq s^T \text{ with } \gamma(z_0, \hat{s}^T) \in Z^F.\end{array} \tag{5.4}$$

That is, projecting strings in $L(T)$ to their first event component leads to a string in $L(G)$ ((5.2)) and projecting a string that leads to a faulty state in T to its first event component leads to a faulty string in $L(G) \setminus K$ ((5.3)). Finally, all strings that lead to indeterminate cycles in T are projected to strings that are not codiagnosable ((5.4)).

In addition, we introduce an efficient algorithm to project the strings in $L(T)$ to their first event component according to the map c_G. To this end, we introduce a relation $\sim_G \subseteq Z \times Z$ between the states of the testing automaton $T = (Z, \Sigma^{\mathtt{T}}, \gamma, z_0)$ as follows. Let

$$\hat{\Sigma}^T = \{\sigma^T \in \Sigma^T | \sigma^T = (\sigma, -, -) \text{ with } \sigma \neq \epsilon\}$$

be the set of events in Σ^T such that the first event component is not equal to ϵ. For $z_1, z_2 \in Z$, we define $\sim_G$ such that

1. $\gamma(z_1, \sigma^{\mathrm{T}}) = z_2$ or $\gamma(z_2, \sigma^{\mathrm{T}}) = z_1$ for some $\sigma^{\mathrm{T}} \in (\Sigma^{\mathrm{T}} \setminus \hat{\Sigma}^{\mathrm{T}}) \Rightarrow z_1 \sim_G z_2$
2. $z_1 = (z_1', \sigma^{\mathrm{T}})$ and $z_2 = (z_2', \sigma^{\mathrm{T}})$ for $\sigma^{\mathrm{T}} \in \hat{\Sigma}^{\mathrm{T}}$ and $z_1' \sim_G z_2' \Rightarrow z_1 \sim_G z_2$.

The relation $\sim_G$ is adopted from [11, 15], where it is introduced in a different context. It holds that $\sim_G$ defines an equivalence relation between the states of T and it is possible to define the quotient automaton of T with respect to $\sim_G$. Let E^T denote the set of equivalence classes according to $\sim_G$. Then, the quotient automaton $T/\sim_G = (E^T, \Sigma, \xi, e_0^T)$ is defined such that

$$\begin{aligned} & e_0^T \text{ is the equivalence class that contains } z_0 \\ \forall e \in E^T \text{ and } \sigma \in \Sigma : \xi(e, \sigma) = e' & \text{ if there are } z \in e, z' \in e', \sigma^T = (\sigma, -, -) \in \Sigma^T \\ & \text{such that } z' = \gamma(z, \sigma^T) \end{aligned}$$

That is, each state of $T/\sim_G$ represents an equivalence class and the transitions of $T/\sim_G$ characterize how these states are connected by transitions with events in $\hat{\Sigma}^T$.

In figure-5.5 a small part of a testing automaton is given with the set $\hat{\Sigma}^T = \{$(a,dont_care,dont_care), (c,dont_care,dont_care)$\}$. That is, the equivalence classes $\{1, 3\}$, $\{2, 4, 5\}$ and $\{6\}$ are found. Accordingly, the quotient automaton $T/\sim_G$ is shown in figure-5.6.

Using the above observations and the construction of the quotient automation, we propose the following algorithm to determine the non-codiagnosable sublanguage of $L(G)$.

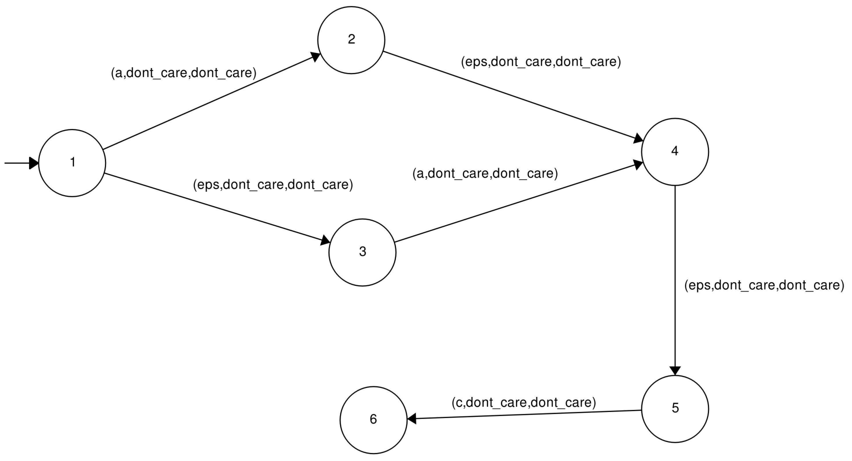
1
2
3
4
5
6
(a,dont_care,dont_care)
(eps,dont_care,dont_care)
(eps,dont_care,dont_care)
(a,dont_care,dont_care)
(eps,dont_care,dont_care)
(c,dont_care,dont_care)

Figure 5.5: Testing Automaton Part Before Step-7

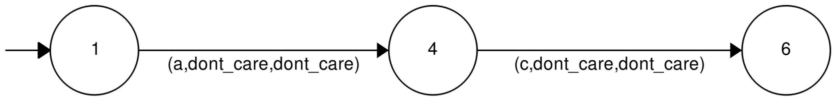

Figure 5.6: Testing Automaton Part After Step-7

Algorithm 2

Input: *G, C, M_1, M_2*

Procedure:

1. *Introduce selfloops for deadlock states in G*
2. *Construct the testing automaton T*
3. *Determine the indeterminate cycles in T and find $\mathcal{C}^T$*
4. *Mark all states in $\mathcal{C}^T$*
5. *Make T trim. The resulting automaton is called $\hat{T}$*
6. *Compute the quotient automaton $\hat{T}/\sim_G$*
7. *Compute $K \cup L(\hat{T}/\sim_G)$. The resulting automaton is called G^{non}.*

Return: *Non-codiagnosable sublanguage of $L(G)$ as $L^{\mathtt{non}} = L(G^{non})$*

In libFAUDES, the non-codiagnosable sublanguage is constructed with the function *NonCoDiagPart* according to Algorithm 2.

```
void NonCoDiagPart (const System& Gen, const Generator& Spec,
const EventSet& ObsEvent1, const EventSet& ObsEvent2,
Generator& SupremalCoDiagPart)
```

In this function the inputs are the plant automaton *Gen*, the specification automaton *Spec* with the local observation masks *ObsEvent1* and *ObsEvent2*. The function produces the *NonCoDiagPart automaton* G^{non} as described above.

We next illustrate the construction by an example. In this example, the plant automaton G is given in figure-3.5, the specification automaton C is given in figure-3.6 and the observation masks are given in table-4.1.

- The first step to build G^{non} is constructing a deadlock free generator. As described in [13], Qui and Kumar suggest a trick to obtain a deadlock free generator by introducing ϵ-selfloops. However, as is shown in Section 4.3, while constructing the testing automaton we observe that this approach is not correct. If we add ϵ-selfloops for deadlock states, it turns out that all indeterminate cycles that are caused by codiagnosability violations in deadlock states are missed. Hence, as suggested in Section 4.3, we introduce selfloops with new unobservable events. This procedure is illustrated in figure-5.7.

  This explanation goes to the section where we explain our correction. Because, as stated in equation-(4.7) testing automaton does not include any event, which three components are equal to ϵ. Therefore, we add new defined events. Generator seen in figure-3.5 has a deadlock state, *state-10*, and we construct automaton in figure-5.7.

- The second step is constructing the testing automaton by using *void TestingAutomaton(const System& Gen, const Generator& Spec,const EventSet& ObsEvent1,const EventSet& ObsEvent2,Generator& Testing).* Details of this function are given in section-4.3. The testing automaton for this example is shown in figure-4.16.

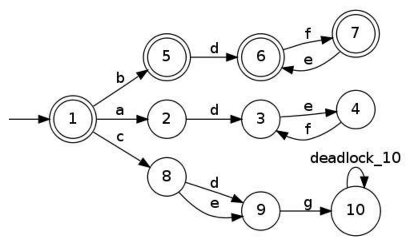

Figure 5.7: Generator without deadlocking states

- For the third step, [13] Qui and Kumar explain that indeterminate cycles have to be found. As described above, we first determine the faulty part of the testing automaton by deleting all non-faulty states as shown in figure-5.8. Then, the strongly connected components in this automaton are computed using the function *bool ComputeSCC (const Generator &rGen, const SccFilter &rFilter,std::list <StateSet> &rSccList, StateSet &rRoots).*

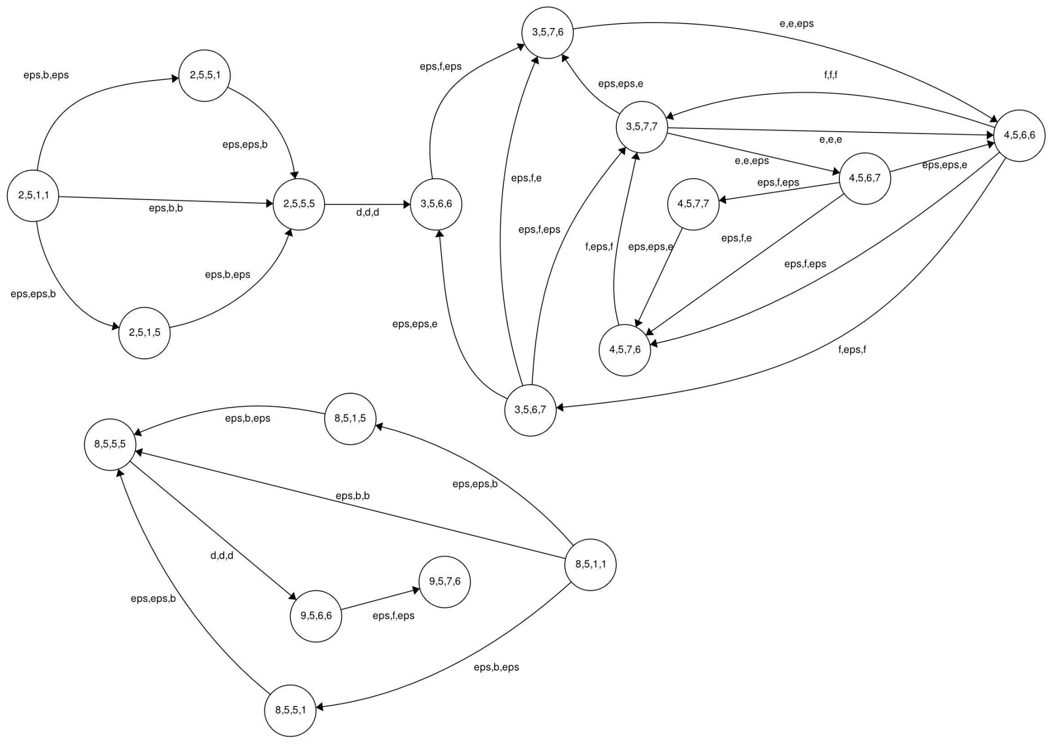

Figure 5.8: Testing Automaton with only Faulty States

- The fourth step is marking indeterminate cycles that were found in step four in the testing automaton. All states in indeterminate cycles are marked by using *void SetMarkedState(Idx index)*. The result for this example is shown in figure-5.9.

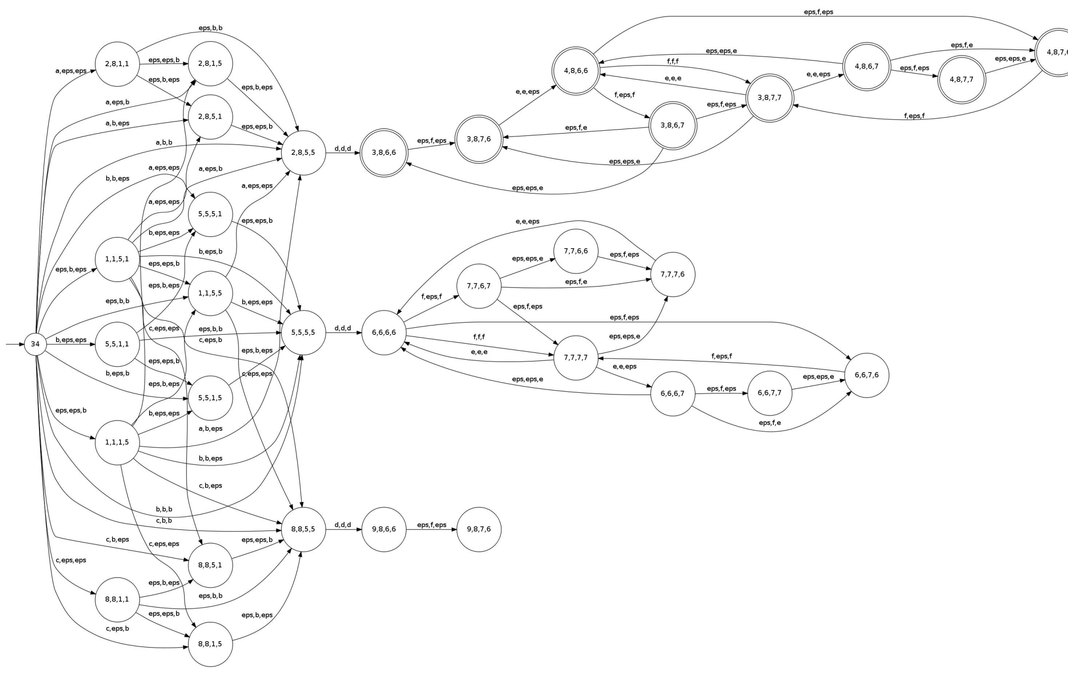

Figure 5.9: Indeterminate cycles marked in Testing Automaton

- Up to this point testing automaton is constructed and all indeterminate cycles are marked. As defined in [13] indeterminate cycles violate codiagnosability. Therefore, a new testing automaton $\hat{T}$ including only indeterminate cycles and strings reaching to indeterminate cycles from the initial state, is created. The function *bool Trim()* in libFAUDES is used to compute this testing automaton. The output of the trim function is shown in figure-5.10.

- The quotient automaton $\hat{T}/\sim_G$ is computed from $\hat{T}$. The result of this step is shown in figure-5.11.

- In section 5.1, we identify noncodiagnosable part as $L^{non} = K \cup (L(G) \setminus L^{co})$. Therefore the last step computes the union of K and the result of the previous step. The function *void LanguageUnion(Generator& G1, Generator G2, Generator G2)* is used for this purpose. $L^{non} = K \cup (L(G) \setminus L^{co})$ for this example is shown in figure-5.12. As seen in this figure, noncodiagnosable part is not equal to neither G generator nor K specification. In other

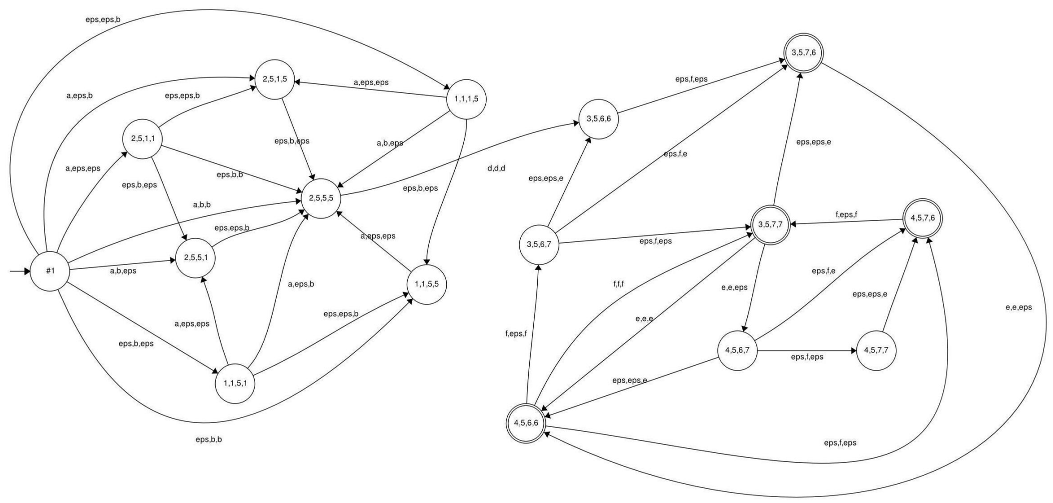

Figure 5.10: Trimmed Testing Automaton

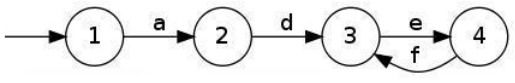

Figure 5.11: $(L(G) \setminus L^{\mathtt{co}})$

words, this automaton can not be diagnosed by decentralized diagnoser and there is no need to diagnose whole system by centralized diagnoser. The part of generator given in figure-5.12 can be diagnosed by centralized diagnoser and the remaining part can be diagnosed by decentralized diagnoser.

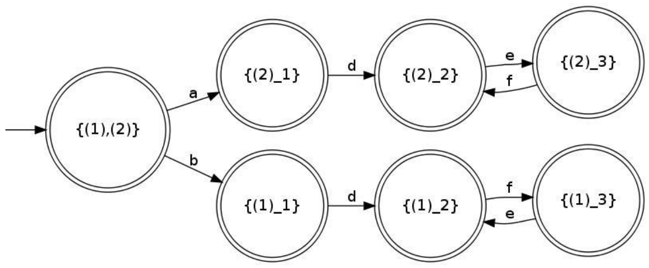

Figure 5.12: $L^{\mathtt{non}} = K \cup (L(G) \setminus L^{\mathtt{co}})$

We next provide several examples in order to further illustrate the computation of G^{non}.

In the first example, we know from before that the language generated by G given in figure-4.17 is codiagnosable with respect to specification language K shown in figure-4.18 and the local observation masks given in table-4.2 where a and f events are unobservable. The testing automaton with only faulty states is shown in figure-4.19. We expect that *NonCoDiagPart* function produces non-codiagnosable part of G given in figure-4.17. The result is in figure-5.13. This automaton includes only the specification automaton because codiagnosability is fulfilled.

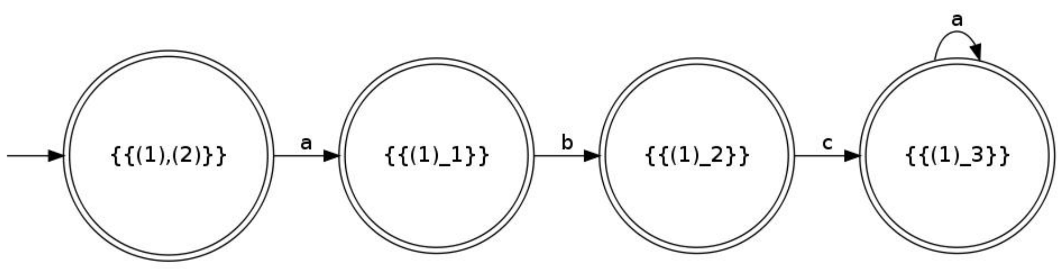

Figure 5.13: $L^{\mathtt{non}} = K \cup (L(G) \setminus L^{\mathtt{co}})$ where $(L(G) \setminus L^{\mathtt{co}}) = \emptyset$

In the second example, the language generated by G given in figure-4.20 is not codiagnosable with respect to the specification K shown in figure-4.21 and the local observation masks given in table-4.3 where s and f events are unobservable. In this case, the testing automaton produced from G,K,M_1 and M_2 includes indeterminate cycles. These cycles are given in figure-4.22. As stated in section-4.3.3, G is not codiagnosable, hence it is expected that the function *NonCoDiagPart* creates the plant automaton itself. The result is shown in figure-5.14

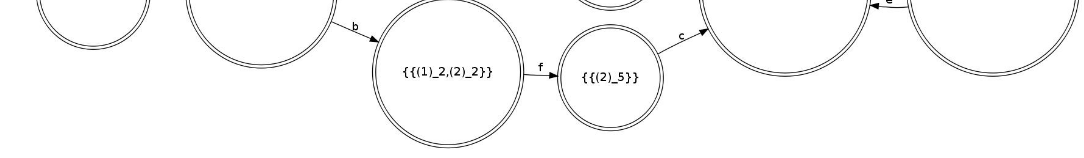

Figure 5.14: $L^{\mathtt{non}} = K \cup (L(G) \setminus L^{\mathtt{co}})$ where $L^{\mathtt{non}} = G$

In the third example, generator G and specification K has unobservable cycles, consisting of only unobservable events. Let G be automaton given in figure-5.15, K be automaton shown in figure-5.16 and local observation masks be given in table-5.1 where a, b, c and g are unobservable events. This generator is codiagnosable, because d is observable with both mask functions and the occurrence of event-d means that the string is faulty, and if there is not any d event then, the string is not faulty. L^{non} is shown in figure-5.17.

Table5.1: Local Observation Masks

$\sigma \in \Sigma$	a	b	c	d	e	f	g
$M_1(\sigma)$	ϵ	ϵ	ϵ	D	ϵ	F	ϵ
$M_2(\sigma)$	ϵ	ϵ	ϵ	D	E	ϵ	ϵ

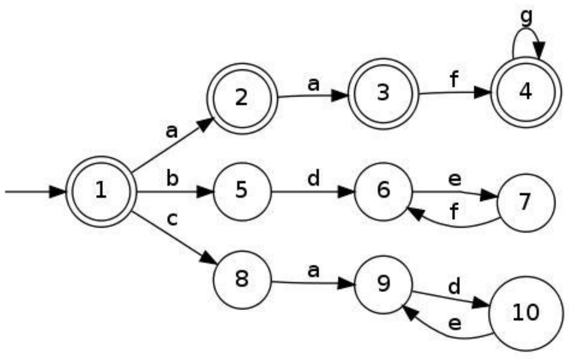

Figure 5.15: Generator, G

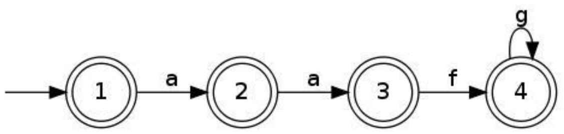

Figure 5.16: Specification including unobservable cycle

In the fourth example, generator G and specification K have deadlocking states. Let G be automaton given in figure-5.18, K be automaton shown in figure-5.19 and local observation masks be given in table-5.2 where a and f are unobservable events and. This generator is diagnosable, b and c events are observable and if c event exists before b event in string, then it is known that this string is faulty.

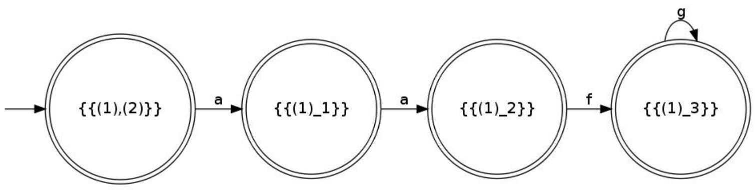

Figure 5.17: $L^{\mathtt{non}} = K \cup (L(G) \setminus L^{\mathtt{co}})$ where $L^{\mathtt{non}} = K$

On the other hand, this generator is not codiagnosable with respect to the local observation masks. It turns out that the noncodiagnosable subpart of this generator is equal to the generator itself. This system includes a deadlock state, so we use the trick explained in Section 4.3.3. The testing automaton includes indeterminate cycles as seen in figure-5.21 and $L^{\mathtt{non}}$ is shown in figure-5.22. We again note that if the trick suggested in [13] is used, the testing automaton consisting of faulty states is constructed as shown in figure-5.20. Although the system is not codiagnosable, this testing automaton has no indeterminate cycles. This shows that the computation in [13] is not correct.

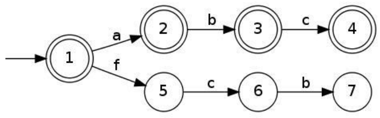

Figure 5.18: Generator including deadlocking state

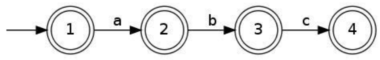

Figure 5.19: Specification automaton

Table5.2: Local Observation Masks

$\sigma \in \Sigma$	a	b	c	f
$M_1(\sigma)$	ϵ	B	ϵ	ϵ
$M_2(\sigma)$	ϵ	ϵ	C	ϵ

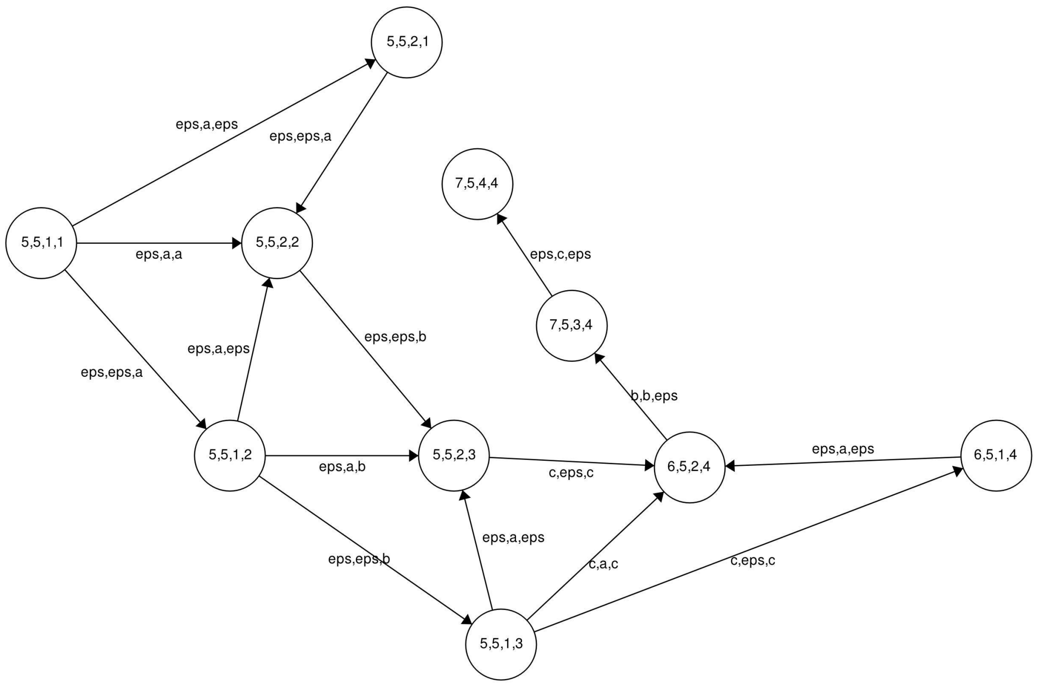

Figure 5.20: Testing Automaton that consists of only Faulty States

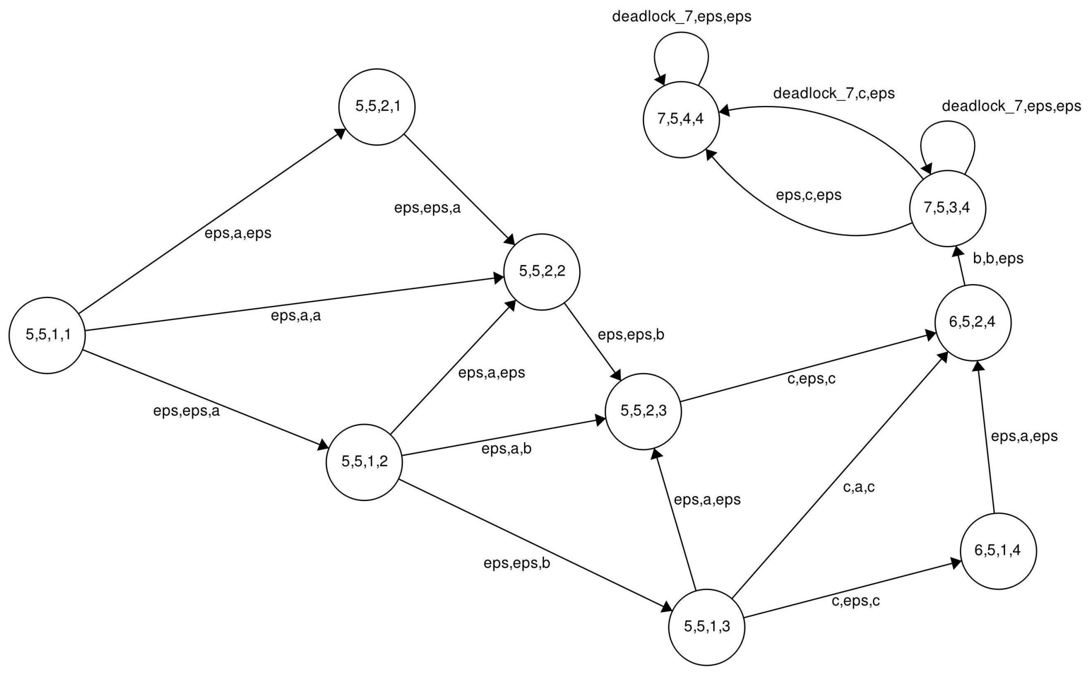

Figure 5.21: Testing Automaton that consists of only Faulty States

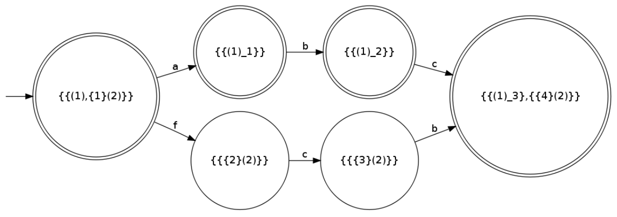

Figure 5.22: $L^{\texttt{non}} = G$

CHAPTER 6

APPLICATION EXAMPLES

In this chapter, combined decentralized and centralized diagnosis is applied to two example systems. In section 6.1, a communication system with three nodes and potential link failures is considered and section 6.2 presents a manufacturing system example.

6.1 Communication Application

We apply the algorithm suggested in this thesis to a communication system example. The system consists of three nodes that are connected by a communication link as illustrated in figure-6.1. Hereby, the connections A-B and C-D can break and hence disrupt the communication among the nodes.

We consider the case of an inquiry of Node 1 that has to be confirmed by Node 2 and 3. After receiving the confirmation, Node 1 will send a command to the other nodes. We next provide model automaton for the communication system, including the potential link failures. To this end, we introduce the fault events $breakA - B$ and $breakC - D$.

6.1.1 Communication System Modeling

The model of Node 1 is given in figure-6.2. In the normal operation, Node 1 generates a request (RC) and sends it on the link A-B ($sendA - B$)). Then, it receives responses from Node 2 and Node 3 ($RC2$ and $RC3$) and send a

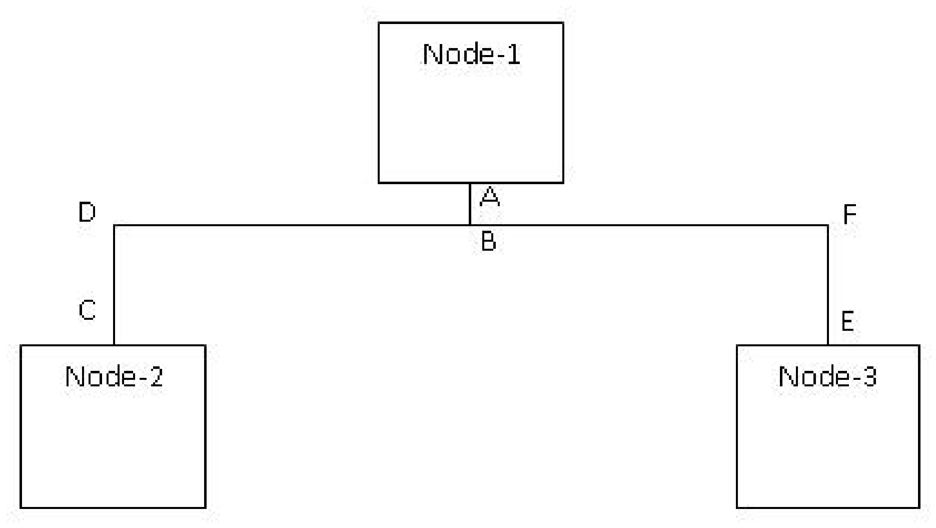

Figure 6.1: Communication system overview.

command (*command*) to complete the interaction. However, the link A-B or C-D can break at any time during the interaction which cannot be directly observed by the Node 1. Nevertheless, after the link failure, too much time will elapse, which will trigger different timers in Node 1 depending on the last observed event of Node 1. For example, if link A-B breaks directly after $sendA-B$ (state 3), no more events will happen in Node 1 and timer3 elapses.

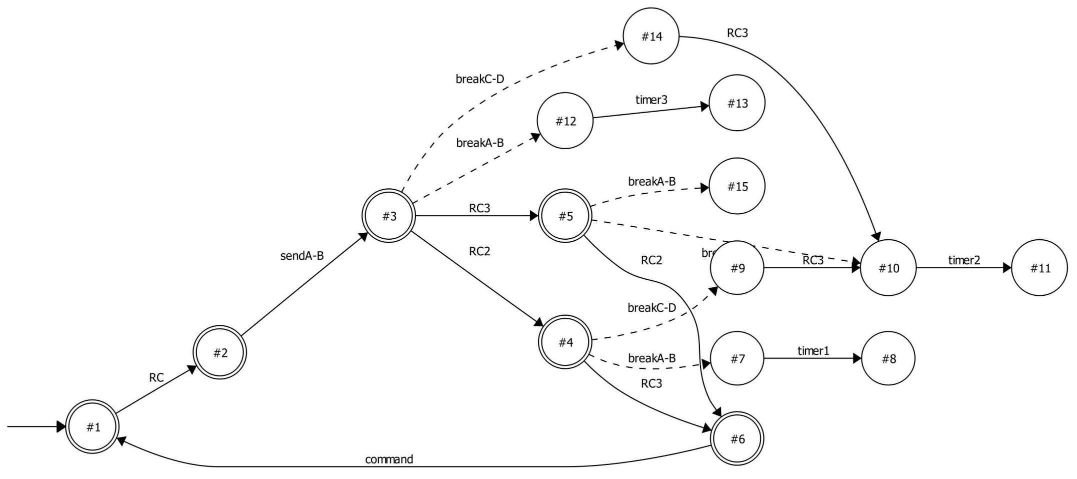

Figure 6.2: Node-1

The model of Node 2 is shown in figure-6.3. In its normal operation, a message is send through the link C-D ($sendC - D$), a response is generated by Node 2

($respC - D$) and the command is received from Node 1 (*command*). Again, different timers are introduced to indicate if too much time elapses during the interaction.

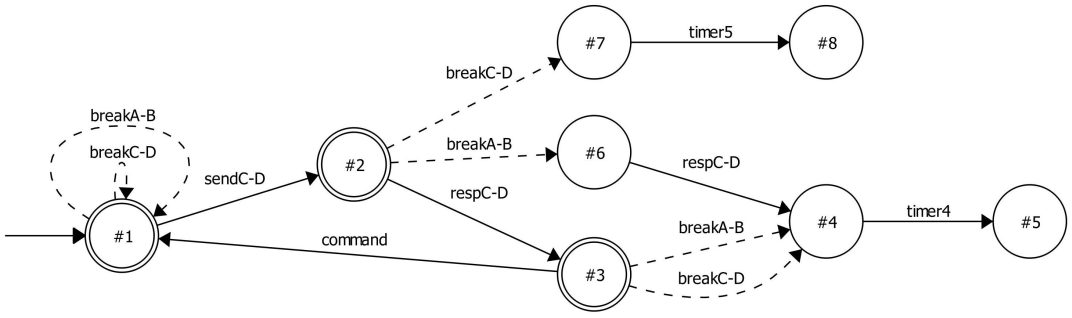

Figure 6.3: Node-2

The model of Node 3 is shown in figure-6.4. A message is send through the link E-F($sendE - F$), Node 3 generates a response ($respE - F$) and the command is received from Node 1 (*command*). In the model of Node 3, there is only one faulty event, *breakA-B*. If this event occurs after a message was send on the link E-F, timer5 will elapse.

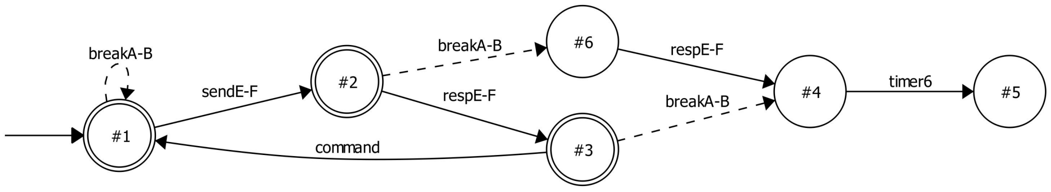

Figure 6.4: Node-3

There are two system models for the link A-B. These models are shown in figure-6.5 and 6.6. In the first model, a message generated by Node 1 is sent on the Link A-B ($SendA-B$) and then forwarded to the link C-D ($SendC-D$) as long as no link breaks. However, in case of a link failure ($breakA-B$ or $breakC-D$), no more transmissions are possible and the automaton model goes to a deadlock state.

In the second model, the communication between Node 1 and Node 3 on link A-B is shown. In this system, there is not any faulty event.

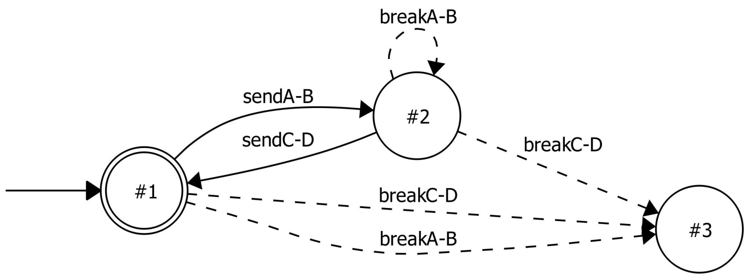

Figure 6.5: A-B Link: Transmission to C-D

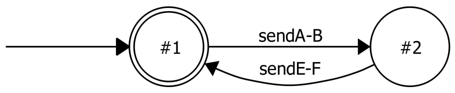

Figure 6.6: A-B Link: Transmission to E-F

The model of link C-D is shown in figure-6.7 (left-hand side). On the link between C and D, there are two types of messages. $repsC-D$ is sent by Node 3 and then forwarded to Node 1 as $RC2$. C-D link may break; if this occurs, the link model goes to a deadlock state and no further communication is possible on that link.

Finally, link E-F is seen in figure-6.7 (right-hand side). This system is analogous to Link C-D, but there are not any faulty events in this link.

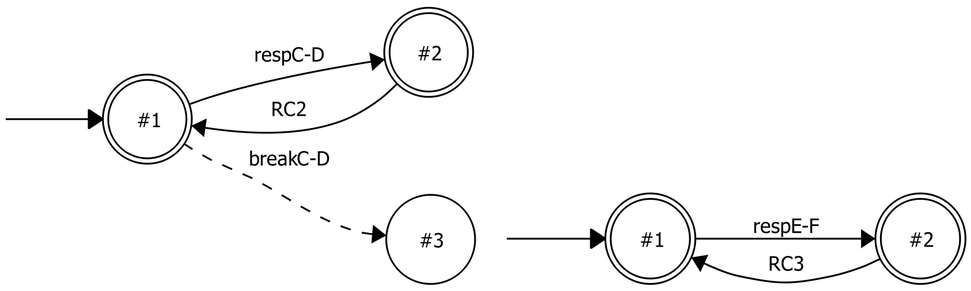

Figure 6.7: C-D Link

The overall plant automaton is given by the synchronous composition of its component automata. The result of this computation is shown in figure 6.8.

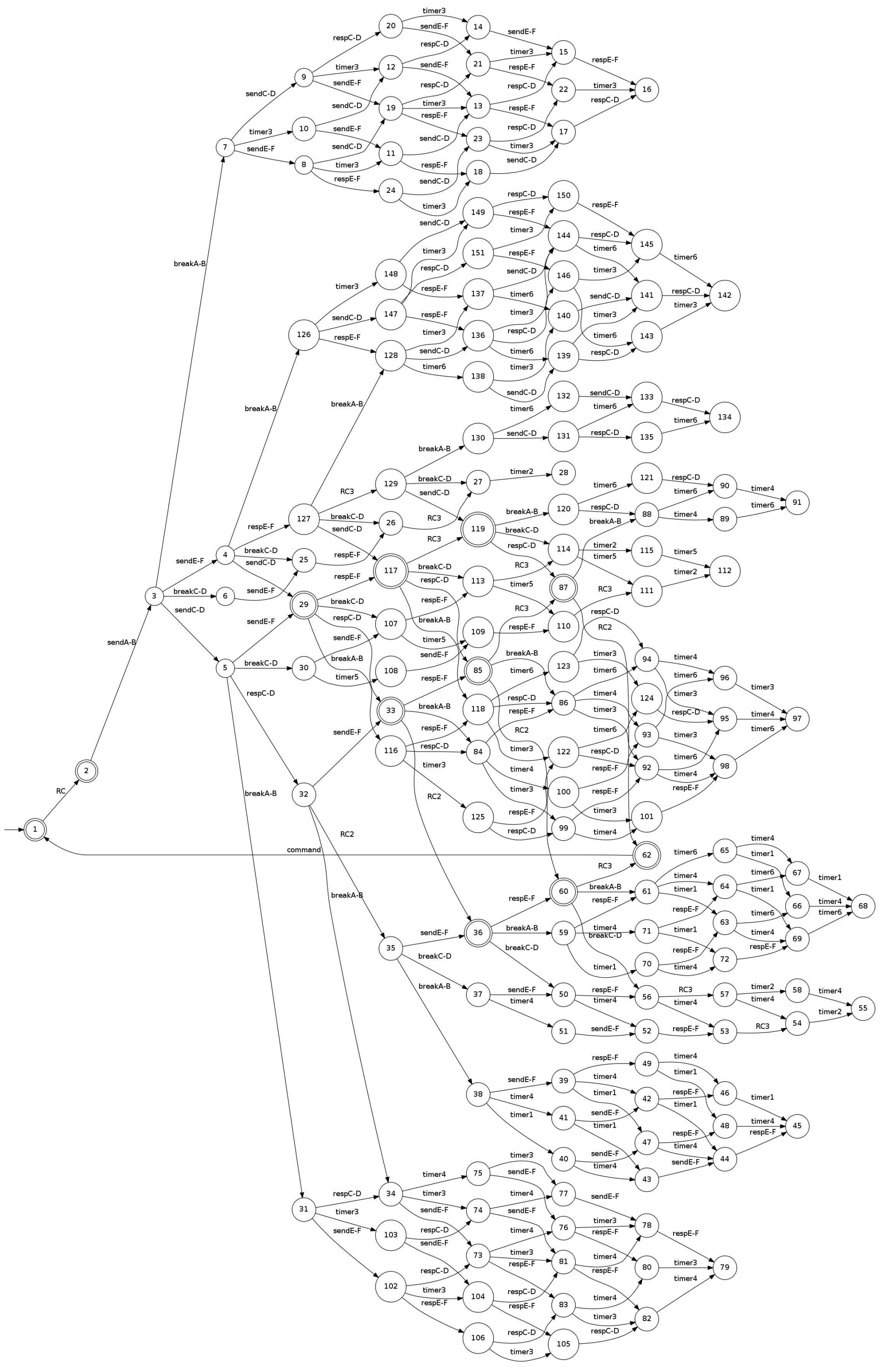

Figure 6.8: Communication Generator

6.1.2 Diagnosing of "$BreakA - B$" Event

We want to focus on the occurrence of a failure of link A-B which is indicated by the event *breakA-B*. Hereby, we assume that all fault events ($breakA - B$ and $breakC - D$) are unobservable. We use a specification automaton that contains all plant strings without the occurrence of $breakA - B$ as shown in figure 6.9.

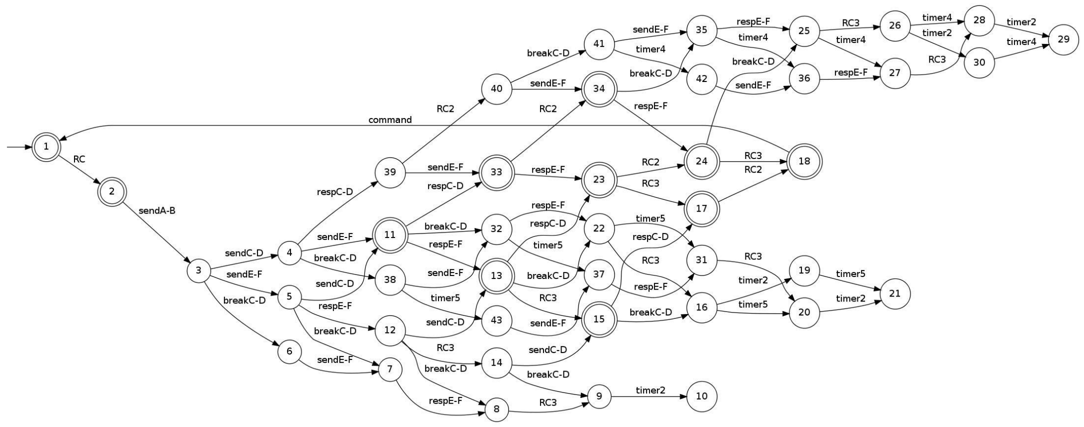

Figure 6.9: Communication Specification Automaton including $breakC - D$ events

In this example, we investigate the case where Node 2 and Node 3 observe the communication system behavior and should decide if a failure of link A-B happened. That is, we use the local observations of Node 2 ($sendC - D$, $respC - D$, $command$, $timer4$, $timer5$) and Node 3 ($sendE - F$, $respE - F$, $command$,$timer6$) to check codiagnosability with respect to the given specification. In this case, it turns out that codiagnosabiltiy is violated but a subset of the potential fault occurrences can be detected. The set of strings that cannot be detected by decentralized diagnosis is shown in figure-6.10.

Analyzing the set of strings that cannot be detected by decentralized diagnosis, it can be seen that the failure of link A-B cannot be detected by Node 2 and Node 3 if it happens before a message is sent to Node 3 ($sendE - F$) in states 3, 5, 32, 35. This is reasonable by intuition, since Node 3 needs to observe that the interaction between the nodes started, which happens by receiving a message from Node 1. Only then, it is possible to detect too much time elapse by timer6 after Node 3 sends its response but does not get a command due to the link

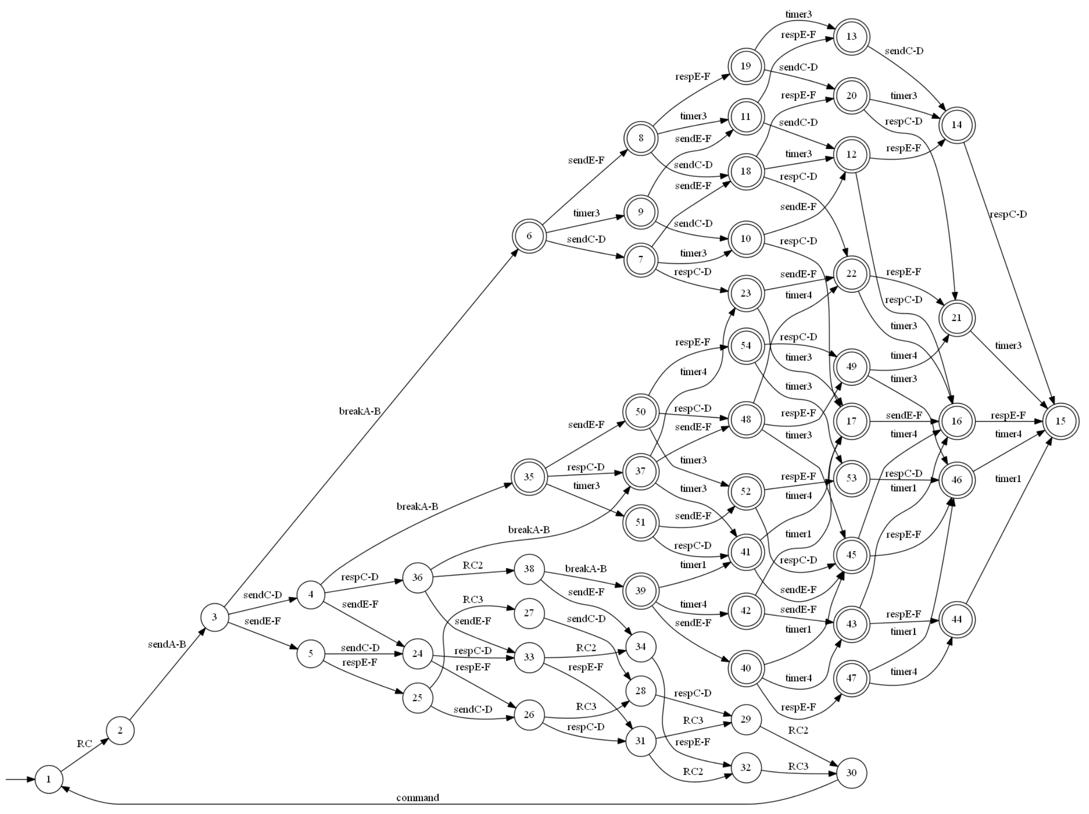

Figure 6.10: Non-codiagnosable strings $L^{non} \setminus K$

failure of A-B. The instances of the link failure A-B before a message is sent to Node 3 have to be detected by centralized diagnosis.

6.1.3 Diagnosing of "$BreakC - D$" Event

In this section, we want to show codiagnosability of *breakC-D* event. Therefore, *breakA-B* event is assumed to be a nonfaulty and unobservable event. If *breakC-D* is the only faulty event in the system, specification automaton becomes as seen in figure-6.11.

We focus on codiagnosability of the occurrence of *breakC-D* with respect to the local observation masks of Node 1 and Node 3. As stated in section-subsec:breakAB, ($sendA - B$,RC,$RC2$,$RC3$,$timer1$,$timer2$,$timer3$,$command$) are observable events in Node 1 and ($sendE - F$, $respE - F$, $command$,$timer6$) are observable events in Node 3. After applying the algorithm suggested in

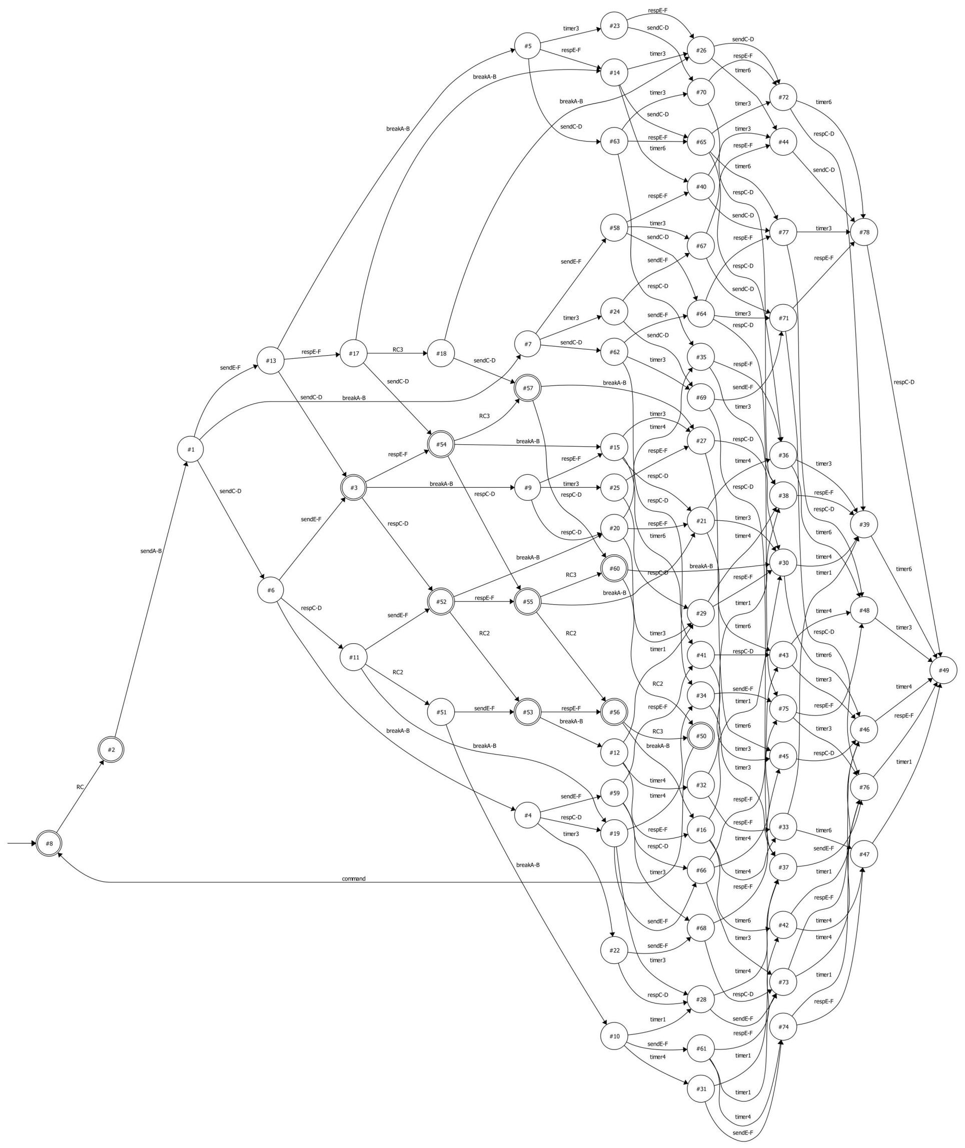
Figure 6.11: Communication Specification Automaton including $breakA - B$

this thesis work, the non-codiagnosable subpart turns out to be equal to the specification automaton. That is, this system is codiagnosable with respect to specification automaton and Node 1, Node 3 observation masks.

6.2 Manufacturing Application

We apply the algorithm suggested in this thesis to a manufacturing system example. The system consists of two main parts, *stack feeder(SF)* and *conveyor belt(C1)* as illustrated in figure-6.12. Hereby, different faults can occur: a product that is put into the stack feeder may not be recognized correctly (*fault* event), a product may stuck between the two system components (*stuck* event) and it may leave the stack feeder in a wrong orientation (*diag* event).

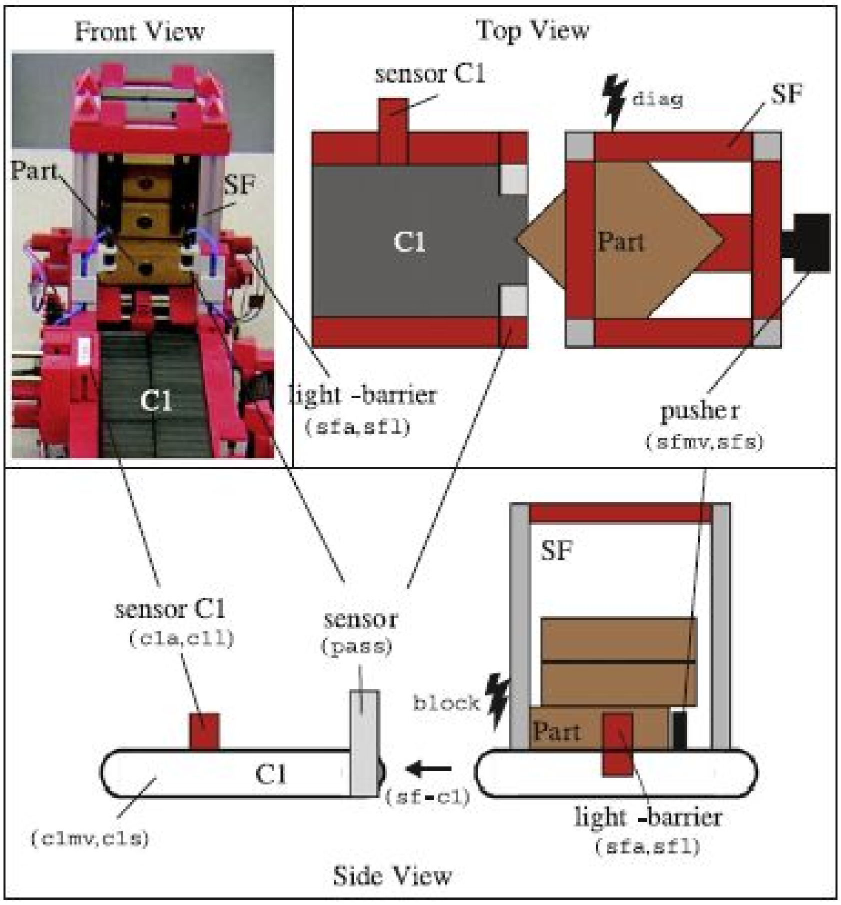

Figure 6.12: Manufacturing Application

The basic operation of the manufacturing system is as follows. After putting a product into the stack feeder, it detects the product and and pushes it to the conveyor belt. This pushing action continues until the product leaves the stack feeder and reaches the conveyor belt. Then, the conveyor belt removes the product from the manufacturing system.

6.2.1 Manufacturing System Modeling

The model of the stack feeder (SF) is given in figure-6.13. In the normal operation, a product of type A or B is detected by SF (sfA, sfB event) and the transition between SF and C1 is initiated ($sf - cl$ event). To this end, the pusher in SF starts to push the product and the product leaves SF and is passed to C1 (*pass* event). However, a product can be stuck at the end of SF (*stuck* event). At this time, no product can not leave SF and move to C1. Hence, the system goes to a deadlocking state. On the other hand, a product can be placed in a wrong orientation and can not leave the SF. SF will assume that new products arrive and try to move this object to C1.

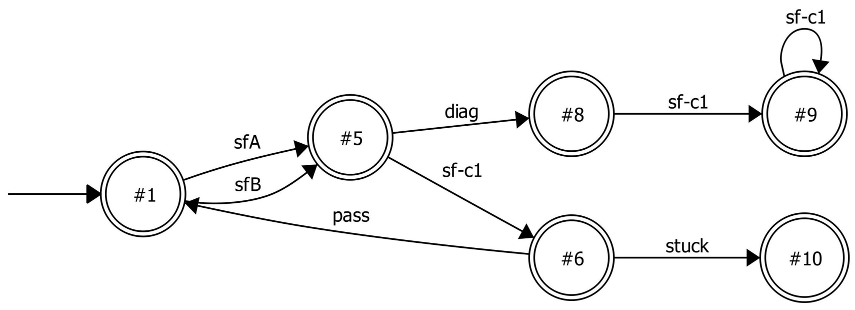

Figure 6.13: Stack Feeder Part

The model of C1 is shown in figure-6.14. In its normal operation, a product comes from SF ($sf - c1$) and is passed to C1 (*pass*). After that, either a product of type A or B reaches C1 ($c1 - A$ or $c1 - B$). However, if the faults *diag* or *stuck* occur, no product arrives at C1 and the timer *timer* elapses.

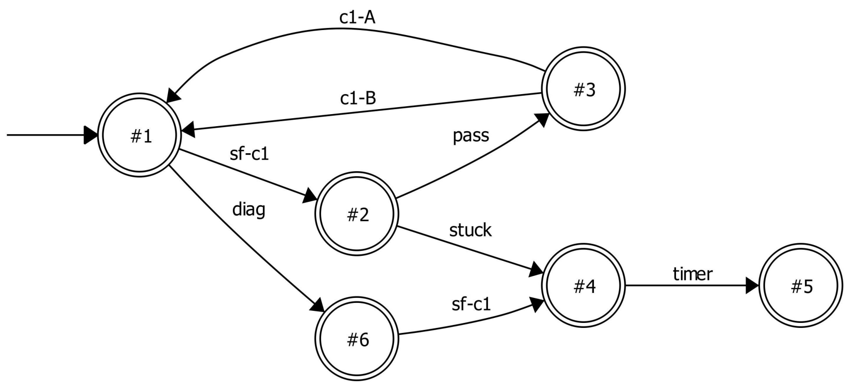

Figure 6.14: Conveyor Belt Part

Last, we model a fault that is related to the detection of the product type. It is possible that a product of type B arrives but is detected as type A. This is indicated by $fault$ in figure 6.15.

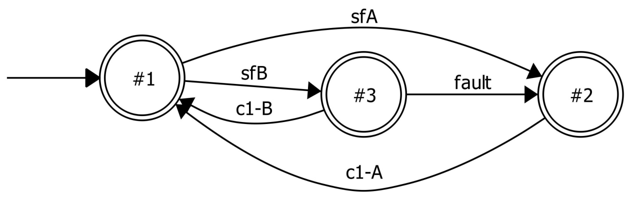

Figure 6.15: Product detection fault

The overall plant automaton is given by the synchronous composition of its component automata. The result of this computation is shown in figure 6.16.

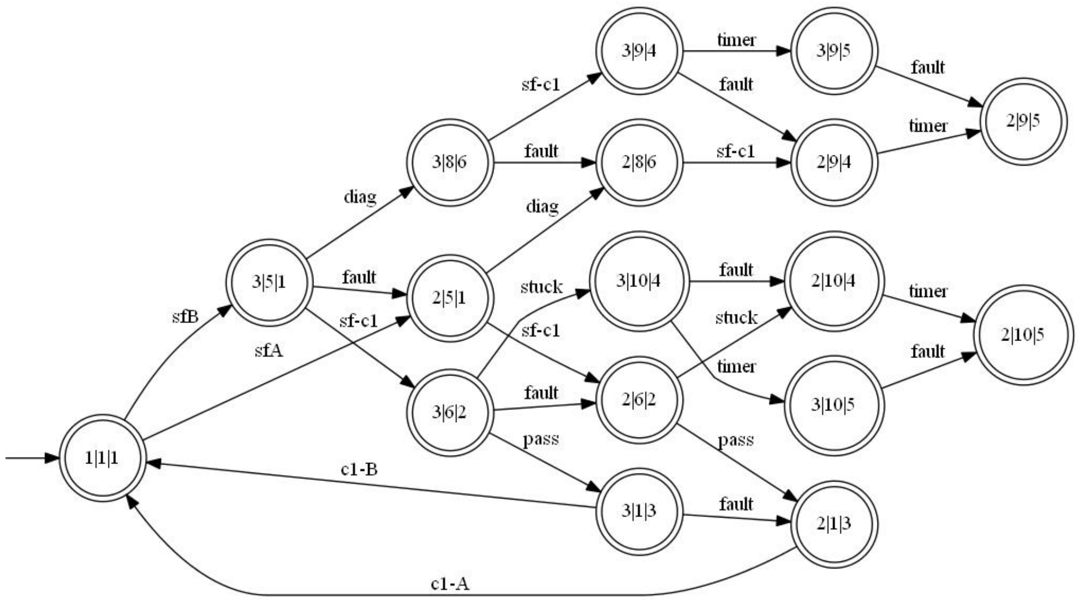

Figure 6.16: Manufacturing System

6.2.2 Diagnosability of Manufacturing System

According to the faults described in the previous section, the specification automaton for the manufacturing system is given in figure 6.17. It describes the correct transport and detection of products in SF and C1.

We next determine the supremal codiagnosable sublanguage for the manufacturing systems using our algorithm. To this end, we compute the non-codiagnosable

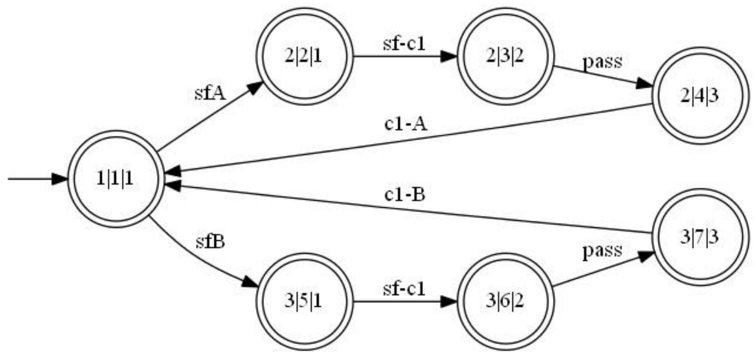

Figure 6.17: Manufacturing Specification Automaton

part of the system which is shown in figure 6.18.

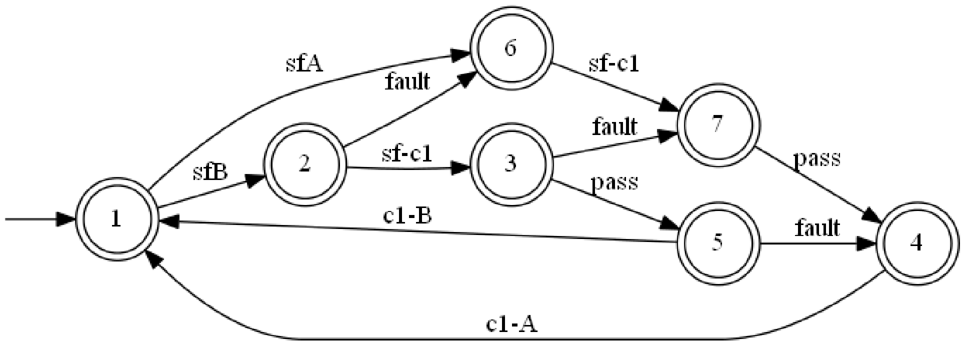

Figure 6.18: Noncodiagnosable Subpart of the Manufacturing System

It turns out that all strings that contain the product detection fault ($fault$) are not codiagnosable, whereas the other faults (*diag* and *stuck*) are codiagnosable. Intuitively, this is not surprising since it is required to know the sequential order of events in SF and C1 in order to diagnose $fault$. However, this sequential order is not known to any of the local diagnosers. Hence, centralized diagnosis is needed to detect $fault$.

CHAPTER 7

CONCLUSION

This thesis proposes a novel fault diagnosis approach for Discrete Event Systems (DES) which combines centralized and decentralized approaches to achieve a correct fault diagnosis with reduced complexity.

The thesis first shows that there is a *supremal codiagnosable sublanguage* that contains all strings that can be diagnosed by decentralized diagnosis. Then a polynomial time algorithm is developed to compute this sublanguage. The developed algorithm together with the verification algorithm for codiagnosability according to [13] are implemented using the software library libFAUDES [10, 9]. The testing automaton is the most important part of the verification algorithm. To build a proper testing automaton, twenty-three cases for adding new transitions and states to the testing automaton are evaluated. Furthermore, a mistake in the original formulation of [13] is identified and corrected. Following the decentralized diagnosis, the diagnosis of the remaining system behavior is carried out by centralized diagnosis.

Finally, the applicability of the combined diagnosis approach is demonstrated by several case studies from manufacturing systems and computer networks.

REFERENCES

[1] A. Benveniste, E. Fabre, S. Haar, and C. Jard. Diagnosis of asynchronous discrete-event systems: a net unfolding approach. *Automatic Control*, 48(5):714–727, 2003.

[2] A. Bouloutas, G. Hart, and M. Schwartz. Simple finite-state fault detectors for communication networks. *Communications*, 40(3):477–479, 1992.

[3] C. Cassandras and S. Lafortune. *Introduction to Discrete Event Systems*. Springer, second edition, 2008.

[4] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. The MIT Press, 1990.

[5] S. Das and L. Holloway. Characterizing a confidence space for discrete event timings for fault monitoring using discrete sensing and actuation signals. *Systems, Man and Cybernetics*, 30(1):52–66, 2000.

[6] R. Debouk, S. Lafortune, and D. Teneketzis. Coordinated decentralized protocols for failure diagnosis of discrete-event systems. *Discrete Event Systems: Theory and Applications*, 10:33–86, 2000.

[7] D. Godbole, J. Lygeros, E. Singh, A. Deshpande, and A. Lindsey. Communication protocols for a fault-tolerant automated highway system. *Control Systems Technology*, 8(5):787–800, 2000.

[8] S. Jiang, Z. Huang, V. Chandra, and R. Kumar. A polynomial algorithm for testing diagnosability of discrete-event systems. *Automatic Control*, 46(8):1318–1321, 2001.

[9] libFAUDES. libFAUDES software library for discrete event systems, 2006–2011.

[10] T. Moor, K. Schmidt, and S. Perk. libfaudes - an open source c++ library for discrete event systems.

[11] R. Morin. Decompositions of asynchronous systems. In *CONCUR '98 Proceedings of the 9th International Conference on Concurrency Theory*, 1998.

[12] Y. Pencolé and M. O. Cordier. A formal framework for the decentralised diagnosis of large scale discrete event systems and its application to telecommunication networks. *Artificial Intelligence*, 164(1-2):121–170, 2005.

[13] W. Qui and R.Kumar. Decentralized failure diagnosis of discrete event systems. *Systems, Man and Cybernetics*, 36(2):384–395, 2006.

[14] R. J. Ramadge and W. M. Wonham. The control of discrete event systems. *Proceedings of the IEEE*, 77(1):81–98, 1989.

[15] S. L. Ricker. A structure for verifying observational properties of decentralized discrete-event systems. In *Electronic Proceedings of the 18th IFAC World Congress*, 2011.

[16] M. Sampath. A hybrid approach to failure diagnosis of industrial systems. In *American Control Conference*, volume 3, pages 2077–2082, 2001.

[17] M. Sampath, R. Sengupta, S. Lafortune, K. Sinnamohideen, and D. Teneketzis. Diagnosability of discrete-event systems. *Automatic Control, IEEE Transactions on*, 40(9):1555–1575, 1995.

[18] K. Schmidt. Abstraction-based failure diagnosis for discrete event systems. *System and Control Letters*, 59:42–47, 2010.

[19] K. Schmidt. Abstraction-based verification of codiagnosability for discrete event systems. *Automatica*, 46:1489–1494, 2010.

[20] K. Schmidt. Verification of modular diagnosability with local specifications for discrete-event systems. *Systems, Man, and Cybernetics: Systems, IEEE Transactions on*, 43(5):1130–1140, 2013.

[21] K. W. Schmidt. Combination of monolithic and decentralized diagnosis of discrete event systems under decentralized observation. *Technical Report, Çankaya University*, 2013.

[22] R. Suu and M. Wonham. Global and local consistencies in distributed fault diagnosis for discrete-event systems. *Automatic Control*, 50(12):1923–1935, 2005.

[23] Y. Wang, S. Yoo, and S. Lafortune. Diagnosis of discrete event systems using decentralized architectures. *Discrete Event Dynamic Systems*, 17(2):233–263, 2007.

[24] S. Yoo and S. Lafortune. Polynomial time verification of diagnosability of partially observed discrete-event systems. *Automatic Control*, 47(9):1491–1495, 2002.

[25] T. Yoo and H.E.Garcia. Diagnosis of behaviors of interest in partially-observed discrete-event systems. *System Control Letters*, 57(12):1023–1029, 2008.

[26] S. H. Zad, R. Kwong, and W. Wonham. Fault diagnosis in discrete-event systems: framework and model reduction. *Automatic Control*, 48(7):1199–1212, 2003.