APPLICATIONS OF FAULT-TOLERANT SUPERVISORY CONTROL FOR DISCRETE EVENT SYSTEMS

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES OF
ÇANKAYA UNIVERSITY

BY
SARMAD NOZAD MAHMOOD

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE
IN
THE DEPARTMENT OF
ELECTRONIC AND COMMUNICATION ENGINEERING

JUNE 2014

Title of the Thesis : **Applications of Fault-tolerant Supervisory Control for Discrete Event Systems**

Submitted by **Sarmad Nozad Mahmood**

Approval of the Graduate School of Natural and Applied Sciences, Çankaya University.

[illegible]

Prof. Dr. Taner ALTUNOK

Director

I certify that this thesis satisfies all the requirements as a thesis for the degree of Master of Science.

[illegible]

Prof. Dr. Celal Zaim ÇİL

Head of Department

This is to certify that I have read this thesis and that in my opinion it is fully adequate, in scope and quality, as a thesis for the degree of Master of Science.

[illegible]

Assoc. Prof. Dr. Klaus Werner SCHMIDT

Supervisor

Examination Date: 02.06.2014

Examining Committee Members

Assistant Prof. Dr. Ulaş BELDEK	(Çankaya Univ.)	[illegible]
Assoc. Prof. Dr. Klaus Werner SCHMIDT	(Çankaya Univ.)	[illegible]
Assoc. Prof. Dr. Ece G. SCHMIDT	(ODTÜ)	[illegible]

STATEMENT OF NON-PLAGIARISM PAGE

I hereby declare that all information in this document has been obtained and presented in accordance with academic rules and ethical conduct. I also declare that, as required by these rules and conduct, I have fully cited and referenced all material and results that are not original to this work.

Name, Last Name	: Sarmad Nozad MAHMOOD
Signature	: [illegible]
Date	: 02.06.2014

ABSTRACT

APPLICATIONS OF FAULT-TOLERANT SUPERVISORY CONTROL FOR DISCRETE EVENT SYSTEMS

MAHMOOD, Sarmad Nozad
M.Sc., Department of Electronics and Communication Engineering
Supervisor: Assoc. Prof. Dr. Klaus Werner SCHMIDT

June 2014, 61 pages

Failure-recovery supervisory control for discrete event systems (DES) is concerned with the recovery from faults that impair the desired system behaviors. Hereby, it is required to detect the occurrence of faults using fault diagnosis and then adapt the system operation such that the system can continue its operation with a potentially degraded performance. In addition, it is generally desired to resume the original system operation after a faulty component is repaired.

As the first contribution of this thesis, a new fault diagnosis method is implemented and evaluated. Different from existing approaches, this method allows detecting the *repeated* occurrence of faults and is particularly useful when considering systems where faults can repeatedly occur after repair. As the second contribution of the thesis, a new method for the fault-recovery supervisory control is developed. Similar to existing approaches, this method assumes that the system follows its *nominal* behavior as long as the system is non-faulty. If a fault occurs, the system should at least obey a degraded specification until the *desired behavior under fault* is achieved in a bounded number of steps. As an extension, the proposed method also allows returning to the nominal behavior after system repair. In addition, our approach is based on the idea of *modular supervisory control* and hence scalable to large-scale systems. To the best of our knowledge, there is no other modular approach for the fault-recovery supervisory control. The applicability of the developed method is demonstrated by a medium-size laboratory model of a manufacturing system.

Keywords: Discrete Event Systems, Supervisory Control, Fault Diagnosis, Fault Recovery, Manufacturing Systems.

ÖZ

AYRIK OLAYLI SİSTEMLER İÇİN HATA KALDIRIR GÖZETİMLİ KONTROL UYGULAMALARI

MAHMOOD, Sarmad Nozad
Yüksek Lisans, Elektronik ve Haberleşme Mühendisliği
Tez Yöneticisi: Doç. Dr. Klaus Werner SCHMIDT

Haziran 2014, 61 sayfa

Ayrık olay sistemlerine (DES) ilişkin arıza çözüm denetleme kontrolü, istenen sistem davranışlarını olumsuz etkileyen arızaların çözülmesiyle ilgilidir. Bu noktada, arıza tanısı kullanarak arızaların oluşumunu tespit etmek ve daha sonra sistemin potansiyel olarak daha düşük bir performansta çalışmaya devam etmesini sağlayacak şekilde sistemin çalışmasını adapte etmek gerekmektedir. Ayrıca, genellikle arızalı bir bileşen tamir edildikten sonra sistemin normal çalışmasına devam etmesi beklenmektedir. Bu tezde ilk katkı olarak, yeni bir arıza tanımlama yöntemi uygulanmış ve değerlendirilmiştir. Bu yöntem mevcut yaklaşımlardan farklı olarak, arızaların yineleyerek meydana gelmesini tespit etmeye olanak tanımaktadır ve özellikle tamirden sonra arızaların yineleyerek meydana geldiği sistemler ele alındığında yararlıdır. Bu tezde ikinci katkı olarak, arıza çözüm denetleme kontrolü için yeni bir yöntem geliştirilmiştir. Mevcut yaklaşımlara benzer şekilde, bu yöntem sistem arızalı olmadığı sürece sistemin tanımlı (nominal) davranışını sürdüreceğini düşünmektedir. Arıza meydana geldiği durumda ise sistem birkaç işlem ile arıza durumunda istenen davranışa ulaşılıncaya kadar en azından düşük performans gerekliliklerine uymalıdır. Ek olarak, önerilen yöntem aynı zamanda sistem onarıldıktan sonra tanımlanan davranışa geri dönmeye de olanak tanımaktadır. Ayrıca, yaklaşımımız modüler denetim kontrolü fikrine dayanmaktadir ve dolayısıyla da büyük ölçekli sistemlere de uygulanabilir. Bildiğimiz kadarıyla, arıza denetim kontrolüne ilişkin başka bir modüler yaklaşım bulunmamaktadır. Geliştirilen yöntemin uygulanabilirliği, bir imalat sisteminin orta büyüklükte bir laboratuvar modeli ile gösterilmiştir.

Anahtar Kelimeler: Ayrık Olaylı Sistemler, Denetleme Kontrol, Arıza Tespiti, Arıza Çözümleme, Üretim Sistemleri.

ACKNOWLEDGMENTS

I would like to express my sincere gratitude to my thesis advisor Assoc. Prof. Dr. Klaus Werner SCHMIDT, who has encouraged and guided me throughout this thesis patiently.

I also would like to express my deepest gratitude to my ill mother for her support and encouragement and her prayers that make me successful and hopeful in the life. To my father whose support makes my way of success.

I also would like to express my sincere gratitude to my uncle Waleed Majeed and my aunt, who supported and encouraged me along my life. To my dearest and lovely sister and brother.

I also thank and appreciate the Scientific and Technological Research Council of Turkey (TÜBİTAK), for providing equipment and financial support (during my thesis). This research was supported by TÜBİTAK [Career Award 110E185].

Finally, I thank Çankaya University precisely Mechatronics Engineering Department, as well as Electronic and Communication Engineering Department, for their support and creating the appropriate requirements of laboratories and equipments necessary during my study and research period.

TABLE OF CONTENTS

LIST OF FIGURES

FIGURES

FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS

CCP	Controlled Convergence Problem
CO	Conveyor Belt
DES	Discrete Events Systems
MA	Manufacturing Machine
RMS	Reconfigurable Manufacturing Systems
RMT	Reconfigurable Machine Tools
RT	Rotary Table
RTS	Rail Transport System
SCT	Supervisory Control Theory
SF	Stack Feeder

CHAPTER 1

INTRODUCTION

Discrete event system (DES) models are used for systems that reside on a *discrete state space* and whose state evolution depends on the occurrence of discrete *events* [1]. Examples for DES are manufacturing systems, transportation systems or communication networks [2, 3, 4, 5, 6]. The supervisory control theory for discrete event systems (DES) as introduced by Ramadge/Wonham [7] generally assumes that the system behavior is correct. That is, the potential occurrence of faults is not taken into account. Such assumption is not justified in practical systems, where faults such as the breakdown of a machine, the failure of a sensor or the breakdown of a communication link are common. In this thesis, mainly fault occurrences in manufacturing systems are considered.

Dealing with faults requires methods for the *fault diagnosis*, *failure recovery* and *system repair*. Fault diagnosis is concerned with detecting and identifying the occurrence of a fault. The existing literature on fault diagnosis focuses on the occurrence of permanent faults such that the system is generally assumed to remain faulty for all times. Various fault diagnosis approaches in different architectures are proposed in this setting [3, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]. Fault diagnosis approaches for permanent faults are not suitable if system repair is considered. In that case, the repeated occurrence of faults is possible. Although there are few approaches that investigate *intermittent faults* [20, 21, 22, 23], they are also not suitable since they do not determine the exact number of fault occurrences. Accordingly, a fault diagnosis approach for detecting each repeated occurrence of a fault is developed in the TÜBİTAK research project "A Formal Framework and Continuous Workflow for the Controller Design, Failure Diagnosis and Failure Recovery of Reconfigurable Manufacturing Systems" [24]. This approach is evaluated by examples and the relevant algorithms are implemented in the `libFAUDES` software library [25] in this thesis.

In industrial applications, the occurrence of a fault usually leads to stopping the operation of a running system for system repair and performing a complicated restart procedure in order to resume the system operation. That is, the occurrence of a fault severely impacts the operation of a system, leading to loss of production quantity and quality. Hence, it is highly desired to find ways to automatically recover from faults after its detection. Hereby, fault recovery is used in the sense that the system operation is continued with a potentially degraded system performance and automatically returning to the correct system operation after repair. The existing literature [26, 27, 28, 29, 30, 31, 32] mostly considers recovery from faults without system repair. Only the work in [33, 34] includes repair in the supervisor design. However, the proposed methods are targeted for a monolithic system model and hence not scalable to large-scale systems.

In this thesis, the monolithic method in [34] is extended to modular failure-recovery supervisory control. That is, we consider systems that are composed of multiple system components. In principle, the method suggests the following system behavior:

- In the nominal case (no fault occurred), the system follows a behavioral specification that describes the desired operation of the system.
- If a fault occurs, a degraded operation of the system is permitted.
- After a fault occurrence, it takes a bounded number of event occurrences until the system starts following the desired operation under fault.
- If the faulty part of the system is repaired, the system operation returns to the nominal behavior in a bounded number of event occurrences.

As the main contribution of the thesis, this behavior is achieved by using two modular supervisors per system component. The first modular supervisor realizes the nominal system behavior and moves the system to the behavior under fault in case a fault happens. The second supervisor realizes the behavior under fault and returns the system to the nominal operation in case of repair. The thesis provides detailed algorithms for the supervisor computation and demonstrates the proposed failure-recovery method using a medium-scale application example. It is further shown that, under certain conditions, the resulting system is able to *contain* the fault in the sense that the fault occurrence and repair is not visible to the outside.

Precisely, the contributions of the thesis are listed as follows:

- A new approach for the fault diagnosis with repeated faults including an algorithm for the diagnosability verification. In this thesis, the algorithms for the diagnosability verification are implemented.

- A new approach for the modular fault-recovery supervisory control. In this thesis, the algorithms for the supervisor design are both developed and implemented in the software library `libfaudes`.

- A case study that demonstrates the applicability of the developed fault-recovery method using a manufacturing system laboratory example.

The remainder of the thesis is organized as follows. We first provide background information on discrete event systems, the concept of formal languages and automata models in Chapter 2. In addition, we discuss supervisory control, fault diagnosis and several relevant operations in that chapter. Our new method for the diagnosis of repeated faults is described and illustrated by examples in Chapter 3. Finally, we develop our modular approach for the failure-recovery supervisory control in Chapter 4. We present all algorithms and apply our method to an manufacturing system example. Conclusions and directions for future work are given in Chapter 5.

CHAPTER 2

BACKGROUND THEORY OF SUPERVISORY CONTROL FOR DISCRETE EVENT SYSTEMS

The chapter gives a background about supervisory control for discrete event systems by outlining the main topics that are needed in this thesis.

2.1. Discrete Event Systems and Languages

The study in this thesis is based on the modeling framework of discrete event systems (DES). DES are systems with a set of discrete states and such that transitions between states are triggered by the occurrence of discrete *events*. Consider a simple example of a push button to describe the behavior of a DES. It is able to perform two different tasks: turning the switch ON or OFF if pushed or released. Hence the push button can be modeled with the two discrete states `pushed` and `released`. Transitions between the states happen with the two possible discrete events `pushON`, `pushOFF`. The occurrence of `pushON` event is possible in state `released` and the state `pushed` is assumed afterwards. Reversely, in the `pushed` state, any transition with `pushOFF` leads back to the `released` state.

The concept of formal languages is used to represent the behavior of DES. Let Σ be the finite set of events of a DES, also called alphabet. A finite sequence of events in Σ is called *string*. The length of a string s, denoted by $|s|$, is given by the number of events in s. ε is the empty string with length $|\varepsilon| = 0$. The set that contains all possibilities of finite strings of events in the finite alphabet Σ including (ε) is denoted as $\Sigma^\star$ (**Kleene Closure**).
In the previous example (push button switch), the alphabet is given by $\Sigma = \{\texttt{pushON}, \texttt{pushOFF}\}$. A possible sequence of events is $s = \texttt{pushON pushOFF pushON}$ with length $|s| = 3$. The Kleene closure of Σ in this case is $\Sigma^\star = \{\varepsilon, \texttt{pushON}, \texttt{pushOFF}, \texttt{pushON pushON}, \texttt{pushON pushOFF}, \texttt{pushOFF pushON}, \texttt{pushOFF pushOFF}, ..\}$.

The concatenation of two strings $s_1 \in \Sigma^\star$ and $s_2 \in \Sigma^\star$ is the strings $s_1 s_2$.If we have a string $s = s_1 s_2$ such that $s_1, s_2 \in \Sigma^\star$, then we say that s_1 is a prefix of s. $L \subseteq \Sigma^\star$ is called a *language* over Σ. Another important term related to the languages is called the *prefix closure*. Prefix closure of L contains all the prefixes of strings in L and is written as $\overline{L}$:

$$\overline{L} = \{s \in \Sigma^\star | \exists t \in \Sigma^\star \text{ such that } st \in L\} \tag{2.1}$$

A language L is prefix closed if it is equal to its prefix closure $L = \overline{L}$.

Another operation on languages is the *natural projection*. *natural projection* keeps all events in an alphabet $\hat{\Sigma}$ and deletes the others $(\Sigma \setminus \hat{\Sigma})$ from any string $s \in \Sigma^\star$. Let $\hat{\Sigma} \subseteq \Sigma$. This natural projection is defined as $p : \Sigma^\star \to \hat{\Sigma}$ with

$$p(\varepsilon) = \varepsilon$$

$$p(\sigma) = \begin{cases} \sigma & \text{if } \sigma \in \hat{\Sigma} \\ \varepsilon & \text{otherwise} \end{cases}$$

$$p(s\sigma) = p(s)p(\sigma) \tag{2.2}$$

In our previous example, consider that we want to take care about the event (`pushON`) of pushing the button on, and we want to project ($s =$ `pushON pushOFF pushON`). Then, we use $\hat{\Sigma} = \{$`pushON`$\}$. The function considers the occurrence of all other events not in $(\hat{\Sigma})$ as invisible. That is, these events are removed from the string $s =$ `pushON pushOFF pushON` and the projected result is $p(s) =$ `pushON pushON`.

2.2. Automata

DES can be modeled by a finite state automaton as follows

$$G = (X, \Sigma, \delta, x_0, X_m) \tag{2.3}$$

where

- X is a finite set of *states*.
- Σ is a finite set of *events*.

- $\delta : X \times \Sigma \to X$ is a *partial transition function*.
- $x_0 \in X$ is the *initial state* (state where the automata starts from).
- $X_m \subseteq X$ is the set of *marked states* (states that the system should reach in order to complete a task).

The behavior of an automaton is represented by the languages $L(G)$ and $L_m(G)$:

- $L(G)$ is the closed language which includes all paths that follow event sequences using the transitions starting from the initial state to any state of G.

$$L(G) = \{s \in \Sigma^\star | \delta(x_0, s) \text{ exists}\} \tag{2.4}$$

- $L_m(G)$ is the marked language which includes all paths that follow event sequences using the transitions starting from the initial state to any marked state of G.

$$L_m(G) = \{s \in L(G) | \delta(x_0, s) \in X_m\} \tag{2.5}$$

An automaton G is considered (*nonblocking*) if it satisfies the following condition:

$$\overline{L_m(G)} = L(G) \tag{2.6}$$

The condition above is fulfilled, if all strings in G can be extended to a marked state. An automaton can be *cyclic* or *acyclic*. A cycle in an automaton is a sequence of states $x_1, x_2, \ldots, x_k$ (k is a natural number) such that ($x_1 = x_k$) and for all ($i = 1, \ldots, k-1$), there exists an event ($\sigma_i \in \Sigma$) such that ($\delta(x_i, \sigma_i) = x_{i+1}$). This means, that a system starts from the state (x_1 of G), passes over the transitions in G then, and then returns to the same state (x_1). Accordingly, an automaton G without cycles, is called acyclic.

There are several characteristics in each automata such as *accessibility*, *coaccessibility* or both of them together as *Trim*.

- *Accessible:* The automaton $G = (X, \Sigma, \delta, x_0, X_m)$ is *accessible*, if all states in X can be reached from the initial state x_0. Formally, we write:

$$\forall x \in X, \exists s \in \Sigma^\star \text{ such that } \delta(x_0, s) = x \tag{2.7}$$

The operation $Acc(G)$ makes G accessible by removing all non-accessible states.

- *Coaccessible:* The automaton $G = (X, \Sigma, \delta, x_0, X_m)$ is *coaccessible*, if it is possible to reach a marked state from any state in X. Formally, we write:

$$\forall x \in X, \exists s \in \Sigma^\star \text{ such that } \delta(x,s) \in X_m \tag{2.8}$$

Hereby, we realize that every Coaccessible automaton is *nonblocking* $\overline{L_m(G)} = L(G)$. The operation $CoAcc(G)$ makes G coaccessible by removing all non-coaccessible states.

- *Trim:* If the accessibility and the coaccessibility conditions are fulfilled in the automaton $G = (X, \Sigma, \delta, x_0, X_m)$. Then, the automaton is *trim*. We use the following operation:

$$Trim(G) = CoAcc(Acc(G)) \tag{2.9}$$

Consider that $G = (X, \Sigma, \delta, x_0, X_\mathrm{m})$ and $G' = (X', \Sigma, \delta', x_0', X_\mathrm{m}')$ are finite state automata. G' is a *subautomaton* of G, if $X' \subseteq X$, $x_0' = x_0$ and for all $x \in X'$ and $\sigma \in \Sigma$, it holds that $\delta'(x,\sigma)! \Rightarrow \delta'(x,\sigma) = \delta(x,\sigma)$. In words, The automaton (G') is extracted from G by removing states and transitions. Then, we write ($G' \sqsubseteq G$) if (G') is a subautomaton of G. G' is a *strict subautomaton* of G if additionally $\delta(x,\sigma) \in X' \Rightarrow \delta'(x,\sigma) = \delta(x,\sigma)$. In words, only states are removed from G to obtain G'.

For clarity, we introduce a simple system automaton $G = (X, \Sigma, \delta, x_0, X_m)$ in Fig. 1.

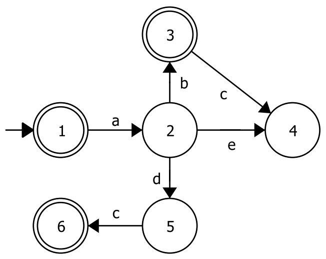


Figure 1: Simple automaton example

G has 6 states and 5 events. The states are ($X = \{1,2,3,4,5,6\}$). Among this set of states there is one initial state $x_0 = 1$ and three marked states $X_m = \{1,3,6\}$. The events are $\Sigma = \{a,b,c,d,e\}$. The transitions relations δ of this simple automaton is represented as follows:

- $\delta(1,a) = 2$, $\delta(2,b) = 3$, $\delta(2,e) = 4$.
- $\delta(2,d) = 5$, $\delta(3,c) = 4$, $\delta(5,c) = 6$.

The closed language for this automata is ($L(G) = \{\varepsilon, a, ae, ab, abc, ad, adc\}$). Then, we can easily find the marked language which is ($L_m(G) = \{\varepsilon, ab, adc\}$). The automaton G is accessible because every state in X, is reachable from the initial state 1. However G is not coaccessible because the state 4 has no transitions to any marked states $X_m = \{1,3,6\}$. Applying *CoAcc*, the state 4 is removed from G. In result, we obtain a strict subautomaton of G. Moreover, it has to be mentioned that G is acyclic.

Discrete event systems are usually modeled using one or more component automata. In the latter case, it is possible to combine all these components in a single system by using the *synchronous composition* operation. This operation synchronizes the different automata on their shared events *same events in each automaton*, whereas the occurrence of the other unshared events are independent from each other.

Consider that we have two different automatas G_1, G_2 , $G_1 = (X_1, \Sigma_1, \delta_1, x_{0,1}, X_{\mathrm{m},1})$ and $G_2 = (X_2, \Sigma_2, \delta_2, x_{0,2}, X_{\mathrm{m},2})$. The synchronous composition is written as:

$$G_1 || G_2 = G_{12} = (X_{12}, \Sigma_{12}, \delta_{12}, x_{0,12}, X_{\mathrm{m},12}) \tag{2.10}$$

Synchronous composition operation states are $X_{12} = X_1 \times X_2$ (*the canonical product of states from X_1 and X_2*), the events are $\Sigma_{12} = \Sigma_1 \cup \Sigma_2$ (*the union of events in Σ_1 and Σ_2*), the initial state is $x_{0,12} = (x_{0,1}, x_{0,2})$, the marked states are $X_{\mathrm{m},12} = X_{\mathrm{m},1} \times X_{\mathrm{m},2}$. The transition makes sure that the events in $\Sigma_1 \cap \Sigma_2$ that are shared by G_1 and G_2 are synchronized. For $(x_1, x_2) \in X_{12}$ and $\sigma \in \Sigma_{12}$:

$$\delta_{12}((x_1,x_2),\sigma) = \begin{cases} (\delta_1(x_1,\sigma), \delta_2(x_2,\sigma) & \text{if } \sigma \in \Sigma_1 \cap \Sigma_2 \wedge \delta_1(x_1,\sigma)! \wedge \delta_2(x_2,\sigma)! \\ (\delta_1(x_1,\sigma), x_2) & \text{if } \sigma \in \Sigma_1 \setminus \Sigma_2 \wedge \delta_1(x_1,\sigma)! \\ (x_1, \delta_2(x_2,\sigma)) & \text{if } \sigma \in \Sigma_2 \setminus \Sigma_1 \wedge \delta_2(x_2,\sigma)! \end{cases} \tag{2.11}$$

2.3. Supervisory Control

Supervisory control was introduced by *Ramadge and Wonham* [7] and is hence called the Ramadge/Wonham framework. The basic supervisory control problem is defined as follows. We write $\Sigma = \Sigma_{\mathrm{c}} \dot{\cup} \Sigma_{\mathrm{u}}$ for *controllable* (Σ_c) and *uncontrollable* (Σ_{u}) events:

- Σ_c represents the set of controllable events: The supervisor can forbid the occurrence of such events whenever is wanted.
- Σ_u represents the set of uncontrollable events: The supervisor can not forbid the occurrence of these kinds of events.

We say that $S = (Q, \Sigma, \nu, q_0, Q_\mathrm{m})$ is a *supervisor* for G with Σ_u if S only disables events in Σ_c. That is, for all $s \in L(G) \cap L(S)$ and $\sigma \in \Sigma_\mathrm{u}$ with $s\sigma \in L(G)$ also $s\sigma \in L(S)$.

A language $K \subseteq L_\mathrm{m}(G)$ is said to be controllable for $L(G)$ and Σ_u if $\overline{K}\Sigma_\mathrm{u} \cap L(G) \subseteq \overline{K}$, and there exists a supervisor S such that $L_\mathrm{m}(G||S) = K$ if and only if K is controllable for $L(G)$ and Σ_u [7]. If K is not controllable for $L(G)$ and Σ_u, the supervisor will implement the *supremal controllable sublanguage* of K. We write:

$$L_m(S||G) = SupC(K, L(G), \Sigma_\mathrm{u}) \tag{2.12}$$

It is ensured that such supervisor is nonblocking and maximally permissive if:

$$SupC(K, L(G), \Sigma_\mathrm{u}) \neq \emptyset \tag{2.13}$$

Considering a plant G and a supervisor S, the closed loop system is obtained by using synchronous composition operation $G||S$. The closed language and the marked language of $G||S$ are obtained as $L(G)||L(S)$ and $L_\mathrm{m}(G)||L_\mathrm{m}(S)$.

The supervisor S is determined to fulfill the desired behavior of the closed-loop system. In discrete event systems the desired behavior usually is represented by an automaton $C = (Y, \Sigma, \beta, y_0, Y_\mathrm{m})$ and $K = L_\mathrm{m}(C)$ is the *specification language*.

A specification is said to be *controllable* with respect to G and $\Sigma_u \subseteq \Sigma$ if it fulfills the condition:

$$\overline{K}\Sigma_u \cap L(G) \subseteq \overline{K} \tag{2.14}$$

- $\overline{K}$: The prefix-closure of (K).
- $\overline{K}\Sigma_u$: The concatenation between the set of all strings that starts with the prefix in $\overline{K}$ and the uncontrollable events in Σ_u.

Consider that($\sigma \in \Sigma_u$). Since no supervisor can disable uncontrollable events, if a specification allows any string ($s \in \overline{K}$) concatenated with an uncontrollable event in the plant ($s\sigma \in L(G)$), then the concatenated string must also belong to the specification ($s\sigma \in \overline{K}$). An important fact in this context is that there exists a nonblocking supervisor S such that $L_{\mathrm{m}}(G||S) = K$ if and only if K is controllable with respect to G and Σ_u.

If the controllability property is violated, the *supremal controllable sublanguage* $K_{sub} \subseteq K$ such that K_{sub} is controllable with respect to G and Σ_u can be obtained. The related operation is written as:

$$SupC(K,G,\Sigma_u) = \bigcup\{K' \subseteq K|K' \tag{2.15}$$

is controllable with respect to G and Σ_u. As result, a nonblocking supervisor S is obtained such that:

$$L_{\mathrm{m}}(G||S) = K_{sub} \tag{2.16}$$

We mentioned that the supervisor enables the largest possible sublanguage of K. Therefore, it is also called (*maximally permissive*).

2.4. State Attraction

Consider an automaton $G = (X,\Sigma,\delta,x_0,X_m)$ and the uncontrollable events (Σ_u). Then, the subset $X' \subseteq X$ is denoted as invariant set in G if there is no transition leaving the states in the subset X':

$$\forall x \in X' \text{ and } \sigma \in \Sigma \text{ it must hold that } \delta(x,\sigma)! \Rightarrow \delta(x,\sigma) \in X'[35] \tag{2.17}$$

Moreover, The set ($X' \subseteq X$) is denoted as (*weakly invariant set*) if all the transitions that leave the states in the subset X' are transitions with controllable events:

$$\forall x \in X' \text{ and } \sigma \in \Sigma_{\mathrm{u}} \text{ it holds that } \delta(x,\sigma)! \Rightarrow \delta(x,\sigma) \in X'[35] \tag{2.18}$$

Definition 1. *Let $A \subseteq X' \subseteq X$ and consider that A,X' are invariant sets in G. A is denoted as a* strong attractor *for X' in G if:*

- *the strict subautomaton of G with the state set $X' \setminus A$ is acyclic*
- *$\forall x \in X'$, there is $u \in \Sigma^\star$ s.t. $\delta(x,u) \in A$*

Briefly, definition 1 means that there must be no arbitrarily long strings outside the state set A and the set A must be reached after a limited number of event occurrences in the system. We also need to mention that the computational complexity of verifying this condition is $\mathscr{O}(|X|+|\Sigma|)$.

Definition 2. *Let $A \subseteq X' \subseteq X$ and consider that A, X' are invariant sets in G. $\Sigma_{\mathrm{u}} \subseteq \sigma$be the set of uncontrollable events. A is denoted as a* weak attractor *for X' in G if there was a state feedback supervisor $S \sqsubseteq G$ s.t. A is a strong attractor for X' in S.*

In words, the set is weak attractor if there exists a supervisor that makes the closed loop system $G||S$ a strong attractor for X' in S.

Moreover, based on the references [35, 36], there exists a supremal subset of X denoted as the set $\Omega_G(A) \subseteq X$ such that A is a weak attractor for $\Omega_G(A)$ in the plant G with the uncontrollable events Σ_{u}. The computational complexity of the algorithm that computes the set $\Omega_G(A)$ is $(\mathscr{O}(|X| \cdot |\Sigma|))$. $|X|$ represents number of state whereas, $|\Sigma|$ represents number of events used in the system, respectively. Finally, we need to mention that the computational complexity of the algorithm that obtains the supervisor ($S \sqsubseteq G$) that makes A strong attractor for $\Omega_G(A)$ in the supervisor S is $\mathscr{O}(|X|^2)$.

2.5. Fault Diagnosis

A fault is an undesirable behavior of a system. Generally, any system can be subjected to different kinds of faults such as:

Transient Fault: The fault can occur one time and does not occur again.

Intermittent fault: This is the most annoying type of faults. The fault can happen, disappear and happen again, etc.

Permanent fault: The fault happens and remains until the system is repaired.

In DES, usually permanent faults are considered. Then, it is assumed that the occurrence of faults cannot be directly observed but faults have to be inferred from the observed system behavior. To this end, we denote the unobservable events as Σ_{uo} and the observable as Σ_o. The alphabet of any system includes both observable and unobservable events such that $\Sigma = \Sigma_o \cup \Sigma_{uo}$. In addition, we use the natural projection $p : \Sigma^\star \to \Sigma_o^\star$.

The fault diagnosis is based on the partially observed DES $G = (X, \Sigma, \delta, x_0)$. $p(L(G))$ is the language that is seen from the plant. A fault is represented by the violation of a given prefix-closed specification language $K = \overline{K} \subseteq L(G)$. That is, the system is fault-free as long as a string $s \in L(G)$ (and any prefix of s) fulfills $s \in K = \overline{K}$, while s is faulty as soon as $s \notin K$. So, the detection is needed, by partial observation through p, if a faulty string in $L(G) \setminus K$ happened.

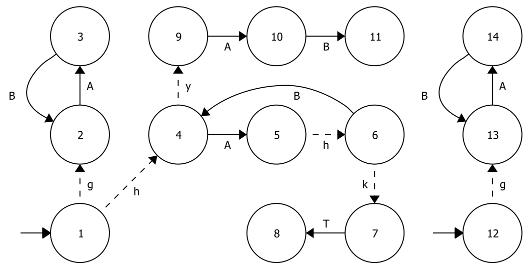


Figure 2: Language specification diagnosability example

Consider Fig. 2 as an example. It shows a plant (left) and a specification (right). We mentioned before that the behavior of the system is realized during the projection $p : \Sigma^{\star} \rightarrow \Sigma_{o}^{\star}$, that maps each event $\sigma \in \Sigma$ to its observation $p(\sigma) \in \Sigma_{o}^{\star}$. Then, in this example if the string s = ($hAh(BAh)^i$ kT) occurred and we found that the projection result of the specification C is (AB) comparing with the projection of s, then the fault is certain because of the difference in the observations. Whereas, if the projection result of the specification C was compared with a new string in this system like s = ($hyAB$), then the fault can not be detected. The following definition of *language-diagnosability* as introduced in [11, 15] formalizes this goal.

Definition 3. *Let G model a DES and let $K = \overline{K} \subseteq L(G)$ be a specification language. K is language-diagnosable w.r.t. G and the natural projection $p : \Sigma^{\star} \rightarrow \Sigma_{o}^{\star}$ if:*

$$\begin{aligned} &(\exists n \in \mathbb{N})(\forall s \in L(G) \setminus K)(\forall st \in L(G), |t| \geq n \text{ or } st \text{ deadlock}) \\ &\Rightarrow (\forall u \in p^{-1}p(st) \cap L(G), u \notin K). \end{aligned} \tag{2.19}$$

The definition above (2.19) considers the faulty strings $s \in L(G) \setminus K$. Then all extended strings with the same projection should be faulty in case a deadlock, also all extended string st with same projection should become faulty if they are longer than a finite detection delay n of events.

It is shown in [11, 15] that language-diagnosability is verified in polynomial time based on G and the specification automaton $C = (Y, \Sigma, \kappa, y_0)$ with $L(C) = K$. If G has p_G

states and q_G transitions, and C has p_C states, then the complexity for this verification is $\mathscr{O}(p_G \cdot q_G^2 \cdot p_C^2)$.

The fault detection is usually realized by a diagnoser automaton that follows the observations from the system. It is a finite state automaton constructed using the system model that is wanted to be put under observation. We introduce a simple system example with its diagnoser in Fig. 3 to identify the normal and the faulty strings in the system according to Definition 3.

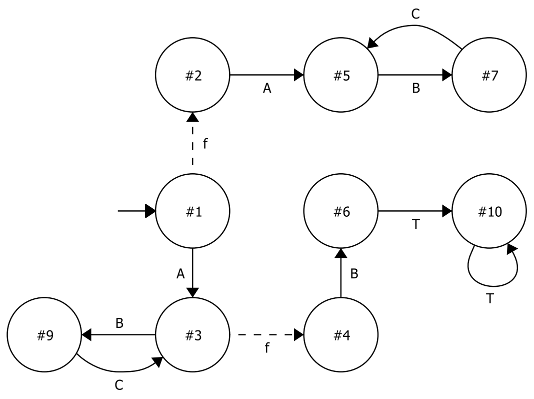


Figure 3: Simple system example G

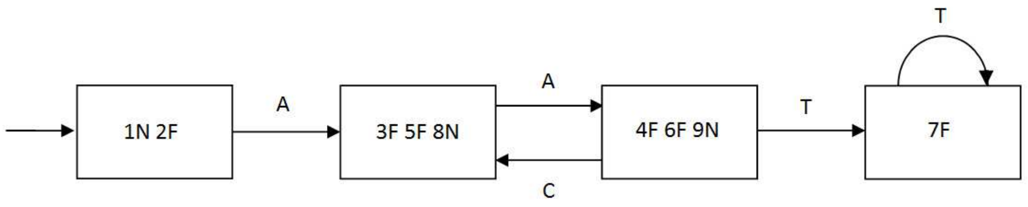


Figure 4: Simple system example Diagnoser D

The example in Fig. 3 shows different strings (faulty and normal). For example the string (fAB) and the stings $(AfBT)$ are faulty strings, whereas the string $(A(BC)^*)$ and $(AB(AB)^*)$ are normal strings. The example violates diagnosability condition, By considering the string:

$$s = f \in L(G) \text{ with } f \in s \tag{2.20}$$

Then, for all $t = A(BC)^{\mathrm{i}}, st \in L(G)$, and $u = A(BC)^{\mathrm{i}} \in p^{-1}p(st) \cap L(G)$ but $f \notin u$, it is not possible to find a bound $n \in N$. So that, the system in Fig. 3 is not diagnosable. This is also observed from the related diagnoser automaton in Fig. 4. There is a loop with the events A and C between the states (3F,5F,8N) and (4F,6F,9N) that contains both fault labels (F) and normal labels (N). This indicates the violation of diagnosability.

2.6. Interleaving Composition

In this thesis, we use the *interleaving composition* as introduced in [37] [page 99]. The operation allows processes with the same alphabet to interleave their operation without synchronizing on shared events. Hence, any activity in the system is the activity of one of the processes. In particular, we need to take a nondeterministic decision in case two processes allow the same event. We write:

$$K_1|||K_2, (K_1 \text{ interleaved with } K_2) \tag{2.21}$$

Formally, we define the interleaving composition as follows:

Definition 4. *Let Σ be an alphabet and $K_1, K_2 \subseteq \Sigma^\star$ be two languages. The* interleaving composition *$K_1|||K_2$ of K_1 and K_2 is defined such that:*

$$\begin{aligned} & s \in K_1|||K_2 \Leftrightarrow s = s_1^1 s_1^2 \cdots s_k^1 s_k^2 \textit{ for some } k \in \mathbb{N} \textit{ and} \\ & s_1^j s_2^j \cdots s_k^j \in K_j \textit{ for } j = 1, 2 \end{aligned} \tag{2.22}$$

2.7. Language Convergence

We further use the notion of *language convergence* [38, 39]. For a string $s \in \Sigma^\star$, we write $\mathrm{suf}_i(s)$ for the string obtained by deleting the first i event from s. Particularly, $\mathrm{suf}_0 = s$ and $\mathrm{suf}_{|s|}(s) = \varepsilon$.

Consider two different languages $M, K \subseteq \Sigma^*$. The language M is said to be *converged* to K, denoted by $K \Leftarrow M$, if there is an integer $n \in \mathbb{N}_0$ such that for each $s \in M$, there exists an $i \leq n$ such that $\mathrm{suf}_i(s) \in K$. The least possible n is named *convergence time*.

The controlled convergence problem which is denoted as (CCP) is introduced in supervisory control for discrete event systems. Consider the plant G over the alphabet Σ, the set of uncontrollable events $\Sigma_\mathrm{u} \subseteq \Sigma$ and the specification $K \subseteq \Sigma^*$. A supervisor S fulfills CCP for G, K and Σ_u if $S||G$ is nonblocking, $S||G$ is controllable for $L(G)$, Σ_u and $K \Leftarrow L_\mathrm{m}(S||G)$. Consider that X is the state set of G and Y is the state set of a recognizer C such that $L_\mathrm{m}(C) = K$. The satisfaction of CCP is decided by an algorithm with complexity $\mathscr{O}(|X|^2 2^{2|Y|})$.

CHAPTER 3

REPEATED FAULT DIAGNOSIS

3.1. Basic Description

The main objective of *fault diagnosis* for DES is diagnosing the systems that are subjected to faults. However, most of the literature is only concerned with permanent faults and does not consider faults that can re-occur. Nevertheless, such faults can very well happen for example after a faulty system component is repaired. This chapter presents a new method for the diagnosis of repeated faults.

We use a state-based model and associate the occurrence of a fault to reaching certain states and presents a polynomial-time algorithm for the verification of the new property of *R-diagnosability*. The algorithm remembers the number of fault occurrences in all strings that have the same observation. We further show the results that the repeated diagnosis function computed (*ComputeRepeatedGo Function*), which is implemented in the diagnosis plug-in of the `libfaudes` software library [25] in this thesis. Last but not least, We mention that automata are considered as four-tuple $G = (X, \Sigma, \delta, x_0)$ in fault diagnosis since the set of marked states is not needed.

3.2. Diagnosability with Repeated Faults

In this section, we model each system as $G = (X, \Sigma, \delta, x_0)$ and the observable events $\Sigma_{\mathrm{o}} \subseteq \Sigma$. The natural projection is $p : \Sigma^{\star} \to \Sigma_{\mathrm{o}}^{\star}$. As in [20], we introduce the *faulty state map* $\psi : X \to \{0,1\}$ such that $\psi(x) = 1$ indicates that a fault happens whenever the state x is reached. Then, the language $K_{\mathrm{f}} \subseteq L(G)$ with:

$$K_{\mathrm{f}} := \{s \in L(G) | \psi(\delta(x_0, s)) = 1\} \tag{3.1}$$

Comprises all strings that are faulty. We further introduce the map $c : X \times \Sigma^{\star} \to \mathbb{N}$ such that:

$$c(x,u) := |\{u' \leq u | \psi(\delta(x,u')) = 1\}|. \tag{3.2}$$

That is, c counts the number of faults in a given string starting from state x. We write $c(s)$ if $x = x_0$.

In this section we illustrates all what is related to diagnosing the systems under recurrent faults occurrences.

Definition 5. *Assume that G is a plant automaton, Σ_o is a set of observable events and $p : \Sigma^{\star} \to \Sigma_o^{\star}$ is a natural projection. Let the maps $\psi : X \to \{0,1\}$ and $c : L(G) \to \mathbb{N}$ be defined as in* (3.1) *and* (3.2). *Then, ψ is denoted as R-diagnosable for G and Σ_o if:*

$$\begin{aligned}&(\exists n \in \mathbb{N})(\forall k \in \mathbb{N})(\forall s \in L(G), c(s) = k)(\forall st \in L(G), |t| \geq n \\ &\textit{or st deadlock})\,[u \in p^{-1}p(st) \cap L(G) \Rightarrow c(u) = k].\end{aligned} \tag{3.3}$$

R-diagnosability as introduced in Definition 5 requires for each string s that indicates $k = c(s)$ faults that the occurrence of k faults can be detected either after a bounded delay or if the plant deadlocks. In particular, Definition 5 implies that the occurrence of the $k+1$st fault is not allowed before the kth fault is detected.

Single faults and multiple fault occurrences were introduced before in Definition 3 and 5. Then, we write the plant G_k for each $k \in \mathbb{N}$ such that:

$$L(G_k) = \{s \in L(G) | c(x_0, s) \leq k\} \tag{3.4}$$

G_k contains all strings $s \in L(G)$ with at most k faults. Also we introduce G_0 such that:

$$L(G_0) = \{s \in L(G) | c(x_0, s) = 0\} \tag{3.5}$$

Then, the recent introduced plant G_0 shows that it contains all strings that are not-faulty. Based on this definition, the following result that was derived in [40] can be used.

Definition 6. *Let G, Σ_o, p, ψ, c be given as in Definition 5. Then, ψ is R-diagnosable for G and p if and only if for each $k \in \mathbb{N}$, $L(G_{k-1})$ is language-diagnosable for G_k and p.*

According to this result, we can easily perform a polynomial time test for each k.

Hence, Definition 6 is useful for plants G where the maximum number of faults is bounded with:

$$\max_{s \in L(G)} \psi(s) = k_{\max} < \infty \tag{3.6}$$

In that case, only $K_{\max}$ classical language diagnosability tests have to be examined. Unfortunately, the test in Definition 6 is useless if no such bound exists.

3.3. Diagnosability Verification for Systems with Recurrent Faults

3.3.1. Certain Number of Faults

We discuss the requirements for the verification of R-diagnosabilty by several examples [40]. In principle, we need to take care whether the fault occurred after the last observation and if the occurrence of such fault is allowed right after the last observation or not.Consider the strings in the systems G_1, G_2, G_3 in the following figures with the set of observable events is $\Sigma_o = \{\alpha, \beta\}$ and the states in gray color represent the faulty states.

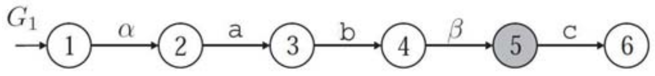


Figure 5: Plants G_1 for illustrating the case of detectable faults

In system G_1 the fault is detected because the faulty state is reachable by the observable event β. Then, any string with projection α is non-faulty, and any string with projection $\alpha\beta$ is considered faulty. Hence, we make sure that the fault is certain after observing the projected string $t = \alpha\beta$. A fault happened before reaching state 6 and right after observing $\alpha\beta$. We denote such fault as *certain*.

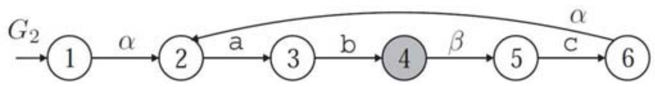


Figure 6: Plants G_2 for illustrating the case of detectable faults

In system G_2 we know from the model that the fault occurred after observing α, but there are different strings that have α projection. For example, the string $s = \alpha\mathtt{ab}$ which is a faulty string and can be divided into two sub strings $s_1' = \alpha, s_2' = \alpha\mathtt{a} < s$. Each string of these has α projection but there is an ambiguity, because s_1 , s_2 are non-faulty whereas the string s is faulty. Hence, it is not possible to detect a fault after $t = \alpha$. Only after observing $\alpha\beta$, the fault is certain. Here, the reason that the fault is not certain immediately is that state 4 was reached but not right after the previous observation α.

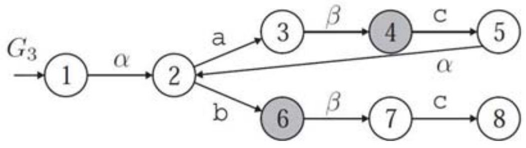


Figure 7: Plants G_3 for illustrating the case of detectable faults

In system G_3, consider the strings $s_1 = \alpha\mathtt{a}$ and $s_2 = \alpha\mathtt{b}$. There is an ambiguity because s_1 is not faulty whereas s_2 is faulty and both of them have the same projection (α). Hence, it is not possible to detect a fault after $t = \alpha$. Or consider different strings $s_1 = \alpha\mathtt{a}\beta$ and $s_2 = \alpha\mathtt{b}\beta$, one fault is certain after observing $\alpha\beta$. but both strings have the same projection ($\alpha\beta$) in this case the fault is not detected because s_1 is faulty and s_2 is not faulty. There is no fault happened when state 3 was reached but one fault happened when state 4 was reached. Hence, one fault is not certain after observing α. Moreover, knowing that a fault happened when state 5 was reached right after observing $\alpha\beta$ and no more fault happens until reaching state 8 shows that a fault is certain after observing $\alpha\beta$.

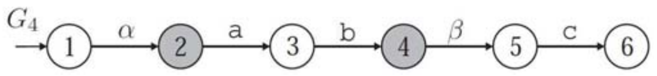


Figure 8: Plants $G4$ for illustrating violation of R-diagnosability

In system G_4, R-diagnosability condition is violated, for the string α because the fault occurs after α, whereas for the string $\alpha\mathtt{ab}$ it is not possible to make a new observation for the new fault. That is, it cannot be decided if one fault or more than one fault happened in the system. For G_4, the classical diagnosability condition is fulfilled because one certain fault occurred after observing α. However, it is not possible to obtain the exact number of faults and R-diagnosability is violated.

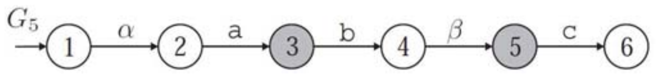


Figure 9: Plants $G5$ for illustrating violation of R-diagnosability

In G_5, R-diagnosability condition is also violated, because there is one fault happened in the string $s_1 = \alpha \mathtt{a}$ whereas for the string $s2 = \alpha \mathtt{ab}$, no fault happened that means, we have both faulty and non-faulty strings with same observation α. Then, it is clear that two faults happened for the string $\alpha \mathtt{ab} \beta$ in the system, but it is not possible to detect the first fault after the string $\alpha \mathtt{ab}$, then only one fault is detected after $\alpha\beta$ observation.

3.3.2. Observer Automaton Construction

The verification procedure for R-diagnosability is composed of two steps. In the first step, we introduce the automaton $S^{\mathrm{o}} = (Q^{\mathrm{o}}, \Sigma_{\mathrm{o}}, \nu^{\mathrm{o}}, Q_0^{\mathrm{o}})$ that detects the observations from G as follows [3, 20, 40].

$$Q^{\mathrm{o}} = \{(x,k,b) \in X \times \{0,1\} \times \{0,1\} \cup \{NR\} | x \in X \text{ and } \exists \sigma \in \Sigma_{\mathrm{o}} \\ \text{ such that } \delta(x,\sigma)! \text{or} \nexists \sigma \in \Sigma \text{ such that } \delta(x,\sigma)!\} \tag{3.7}$$

$$(x,k,b) \in Q_0^{\mathrm{o}} \text{ if and only if } \exists u \in \Sigma_{\mathrm{uo}}^{\star} \text{ such that } x = \delta(x_0,u), \\ c(x_0,u) = k \leq 1, c(x_0,\varepsilon) = b \tag{3.8}$$

$$NR \in Q_0^{\mathrm{o}} \text{ if and only if } \exists u \in \Sigma_{\mathrm{uo}}^{\star} \text{ such that } c(x_0,u) > 1 \tag{3.9}$$

$$\text{For all } q = (x,k,b), q' = (x',k',b') \in Q^{\mathrm{o}} and \sigma \in \Sigma_{\mathrm{o}} \tag{3.10}$$

$$q' \in \nu^{\mathrm{o}}(q,\sigma) \text{ if and only if } \exists u \in \Sigma_{\mathrm{uo}}^{\star} \text{ such that } q' = \delta(q,\sigma u), \\ c(x,\sigma u) = k' \leq 1, c(x,\sigma) = b' \leq 1 \tag{3.11}$$

$$NR \in \nu^{\mathrm{o}}(q,\sigma) \text{ if and only if } \exists u \in \Sigma_{\mathrm{uo}}^{\star} \text{ such that } \delta(x,\sigma u) = x' \text{ and} \\ c(x,\sigma u) > 1 \tag{3.12}$$

Each state of S^{o} states consists of three components (x,k,b). x represents the state name in the system G whether an observable transition was available from x or x was deadlock state, k represents the number of faults after the latest observable transition, b represents the direct or indirect occurrence of a fault after observable transition. Finally, if more than one fault with unobservable event occurred in the system, then NR state is reached in S^{o} from (x,k,b) to the state NR.

The construction algorithm is implemented in the diagnosis plug-in of `libfaudes` as `ComputeRepeatedGo`. The algorithm is designed as follows:

1. Find all states that are reachable from the initial state of G with unobservable

events and count the number of fault occurrences. All those states start form the initial state q_0 of S^o. If there is a string with more than one fault occurrence, NR state becomes reachable from the initial state of S^o.

2. Insert q_0 in the set of waiting states.

3. Take one state q from the waiting states.

4. Find all states x, that are reachable from a state in q by strings σu with one single observable event σ and an unobservable string u. Insert each such state (x,k,b) in the waiting set and add a transition to these states in v^o. Hereby, K counts the number of faults that happen in σu and $b = 1$ if a fault happened directly after σ. Otherwise $b = 0$.

5. Terminate if the waiting set is empty. Otherwise, go to 3.

S^o has at most $|Q^o| \leq |X| \cdot 2 \cdot 2 + 1$ states and at most $|Q^o|^2 |\Sigma_o|$ transitions. Then, the computational complexity of constructing S^o is $O(|X|^2 |\Sigma|)$.

In order to illustrate the construction of S^o, we use `ComputeRepeatedGo` for G_1, G_2, G_3 with the resulting automata in Fig. 10.

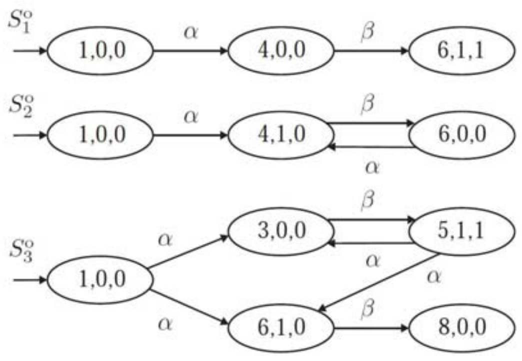


Figure 10: Observer automaton for G_1, G_2, G_3

We further note that our algorithm already shows the violation of R-diagnosability condition for G_4. Precisely, in S_4^o the NR state is reachable from the initial state. This means, that a second fault happened after the last observation but without observing its happening as shown in the following results Fig. 11.

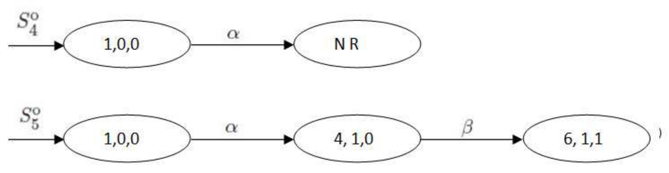


Figure 11: Verifier automaton for G_4, G_5

We used `ComputeRepeatedGo` to test additional systems as shown in the following figures to make sure that the algorithm is working properly. In these figures, the faulty states are indicated by marked states in the plant automata.

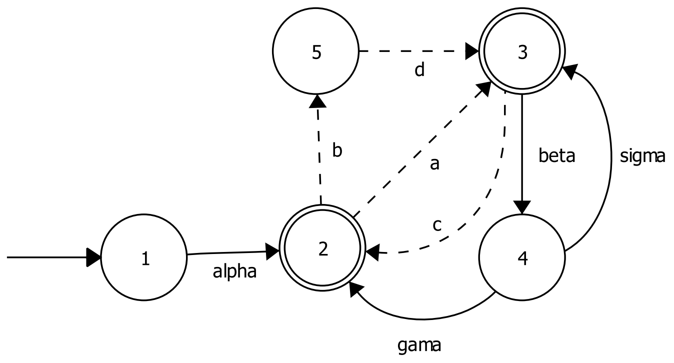


Figure 12: Plant automaton G_6^{input}

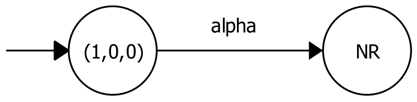


Figure 13: Observer automaton G_6^{result}

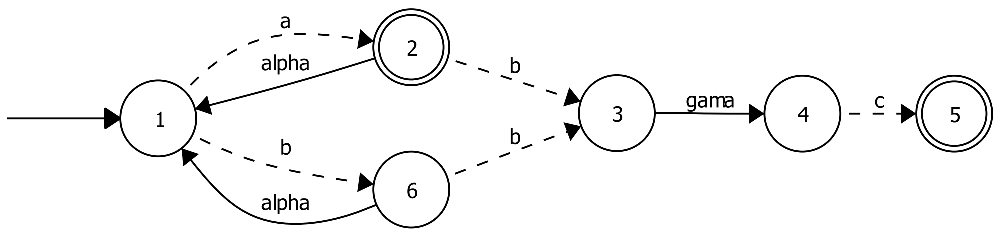


Figure 14: Plant automaton G_7^{input}

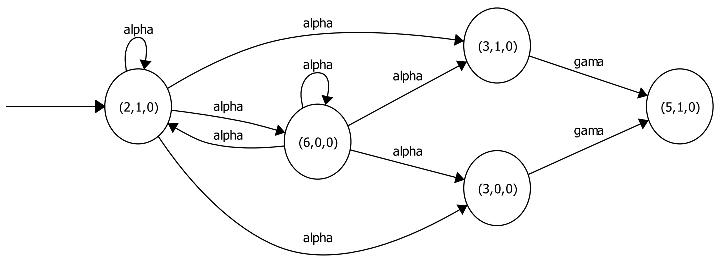


Figure 15: Observer automaton G_7^{result}

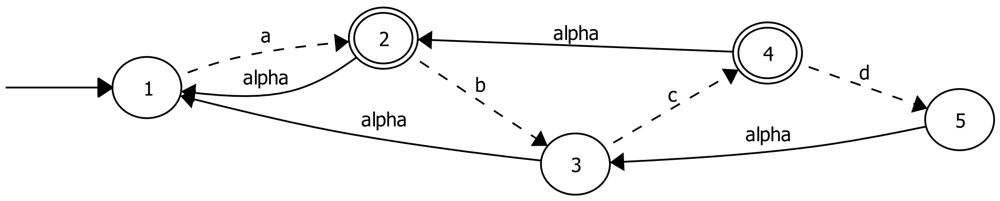


Figure 16: Plant automaton G_8^{input}

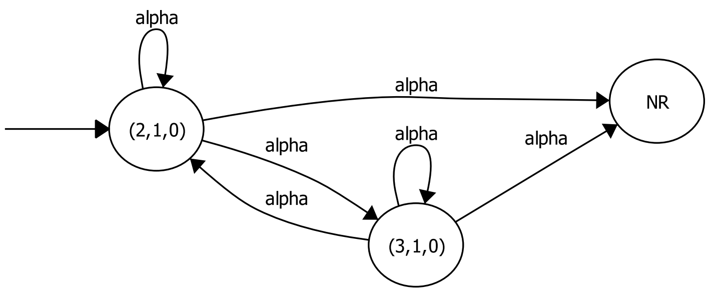


Figure 17: Observer automaton G_8^{result}

Additionally, we intended to use two more systems using `ComputeRepeatedGo` to check the algorithm from different sides to study all error possibilities. The system G_{10} in Fig. 20 and 21 which contains only observable events, fulfills R-diagnosability. In particular, the states components (x,k,b) show that all faults are detected directly because of the observable events that are used in system G_{10}.

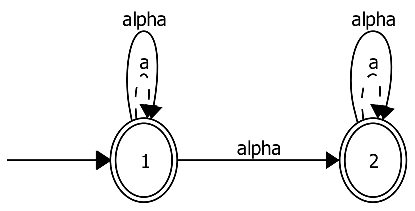


Figure 18: Plant automaton G_9^{input}

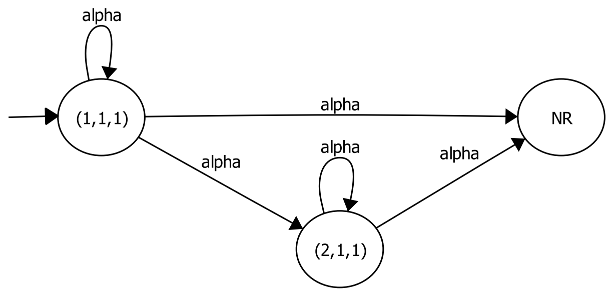


Figure 19: Observer automaton G_9^{result}

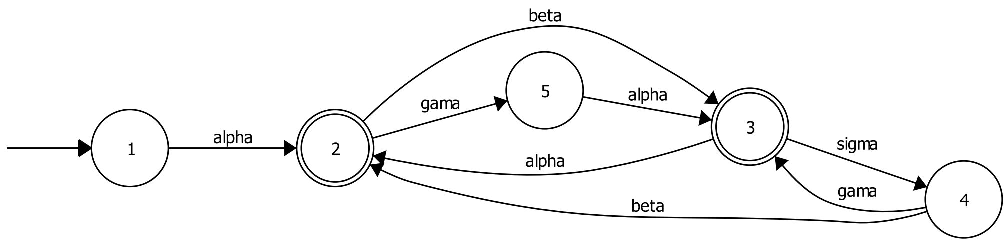


Figure 20: Plant automation G_{10}^{input}

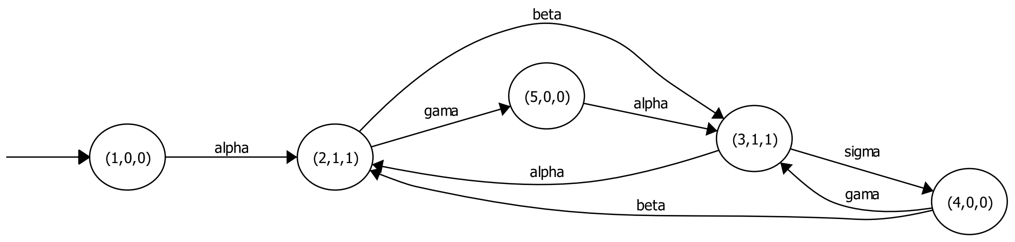


Figure 21: Observer automaton G_{10}^{result}

3.3.3. Verifier Automaton Construction

The observer automaton in the previous section is the basis for R-diagnosability verification. As the next step, we construct a verifier automaton $V = (Q, \Sigma_o, \nu, Q_0)$ by using two copies of S^o automaton. The construction as introduced on [page 10] of [40] is as follows:

$$Q \subseteq Q^o \times \{0,1\} \times Q^o \times \{0,1\} \times \{N,A,F\} \cup \{NR\} \tag{3.13}$$

$$\begin{aligned} Q_0 = \{&(q_0, j, q_0', j', l) | q_0 = (x,k,b), q_0' = (x',k',b') \in Q_0^o, j = k, j' = k' \\ &\text{and } l = N \text{ if } j = j' = 0, l = F \text{ if } b = b' = 1, l = A \text{ otherwise}\} \end{aligned} \tag{3.14}$$

The state of Q consists of the 5-Items (q_1, j_1, q_2, j_2, l). Each item indicates that a change happened in the verifier automaton. These items is represented as follows:

- q_1, q_2 are states of two copies of S^o.
- j_1, j_2 represent the number of faults that happen in the system.
- l represents the state label of the verifier automaton: N (Normal), A (Ambiguous) or F (Fault).

The initial state of the verifier automaton Q_0 contains all state entries q_0, q_0' that are initial states in S^o. Hereby, j, j' is the number of faults that happen in the system for the respective states q_0, q_0' such that the state is labeled as N if it is normal, F if a fault happened in the initial state of the plant ($b = b' = 1$) and A if there exist normal and faulty strings with ε projection.

Then, the transitions that change the verifier state using $\sigma \in \Sigma_o$ events, work as follows:

$$v(q,\sigma) = q' \tag{3.15}$$

For $q = ((x_1,k_1,b_1), j_1, (x_2,k_2,b_2), j_2, l), q' = ((x'_1,k'_1,b'_1), j'_1, (x'_2,k'_2,b'_2), j'_2, l') \in Q$ if :

$$v^o((x_1,k_1,b_1),\sigma) = (x'_1,k'_1,b'_1) \text{ and } v^o((x_2,k_2,b_2),\sigma) = (x'_2,k'_2,b'_2) \tag{3.16}$$

$$j'_1 = j'_2 = 0 \text{ and } l' = N \text{ if } j_1 + k'_1 = j_2 + k'_2 = 0 \tag{3.17}$$

Normal state – no fault in both copies of S^o.

$$j'_1 = 1, j'_2 = 0 \text{ and } l' = A \text{ if } j_1 + k'_1 = 1 \text{ and } j_2 + k'_2 = 0 \tag{3.18}$$

Ambiguous state – the fault happened in the first copy and no fault happened in the second copy of S^o.

$$j'_1 = 0, j'_2 = 1 \text{ and } l' = A \text{ if } j_1 + k'_1 = 0 \text{ and } j_2 + k'_2 = 1 \tag{3.19}$$

Ambiguous state – no fault happened in the first copy and the fault happened in the second copy of S^o.

$$\begin{aligned} &j'_1 = 1, j'_2 = 1 \text{ and } l' = A \text{ if } j_1 + k'_1 = j_2 + k'_2 = 1 \text{ and } k_1 = 0 \wedge b'_1 = 0 \\ &\text{or } k_2 = 0 \wedge b'_2 = 0 \end{aligned} \tag{3.20}$$

Ambiguous state – the fault happened in both the first and the second copy of S^o, but it is said to be ambiguous because no faulty strings with same observation $k_1 = 0 \wedge b'_1 = 0$ or $k_2 = 0 \wedge b'_2 = 0$.

$$\begin{aligned} &j'_1 = 0, j'_2 = 0 \text{ and } l' = F \text{ if } j_1 + k'_1 = j_2 + k'_2 = 1 \text{ and } k_1 = 1 \vee b'_1 = 1 \\ &\text{and } k_2 = 1 \vee b'_2 = 1 \end{aligned} \tag{3.21}$$

Faulty state – the fault happened because ($k_1 = 1 \vee b'_1 = 1$ and $k_2 = 1 \vee b'_2 = 1$) in spite of the number of faults counter in the first and the second copy of S^o was recorded as $j'_1 = j'_2 = 0$ Then, it is reset to start a new fault detection.

$$q' = NR \text{ if } j_1 + k'_1 > 1 \text{ or } j_2 + k'_2 > 1 \tag{3.22}$$

The reachability of NR state means that more than one fault happened with observing only the last observation.

Finally, it is possible to verify R-diagnosability but after introducing another kind of states called deadlock state (Q_d).

$$Q_\mathrm{d} = \{q = ((x_1,k_1,b_1), j_1, (x_2,k_2,b_2), j_2, l) \in Q | \forall\sigma \in \Sigma, \neg\delta(x_1,\sigma)! \\ \text{or } \neg\delta(x_2,\sigma)!\} \tag{3.23}$$

Definition 7. *Let G be a plant, Σ_o be a set of observable events and ψ be a faulty state map. Assume that V is the verifier constructed as described above and Q_d is the set of deadlock states. Then, ψ is R-diagnosable for G and Σ_o if and only if: [40]*

- *The state NR cannot be reached from the initial state Q_0 in the verifier automaton V.*
- *There must be no cycles of states labeled with A in the verifier automaton V.*
- *There must be no deadlock state $q \in Q_\mathrm{d}$ with label A in the verifier automaton V.*

We illustrate the construction of the verifier automaton using $S_1^\mathrm{o}, S_2^\mathrm{o}, S_3^\mathrm{o}$ from before. The results are shown in Fig. 22. All verifiers satisfy the conditions in 7.

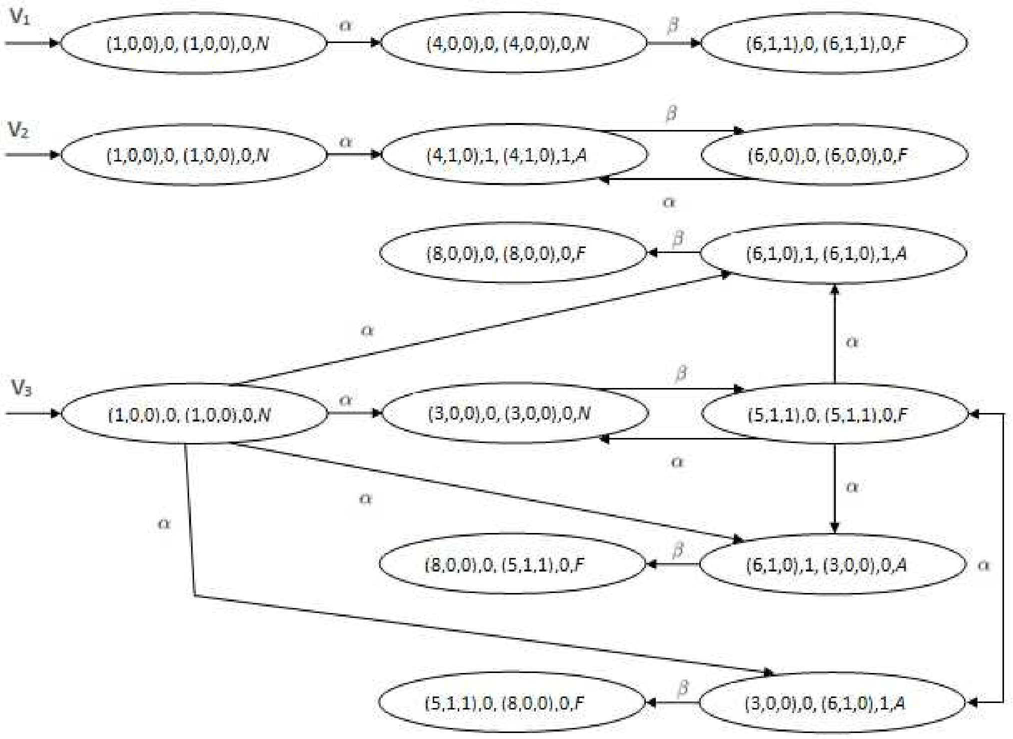


Figure 22: Verifier automata for $S_1^\mathrm{o}, S_2^\mathrm{o}, S_3^\mathrm{o}$

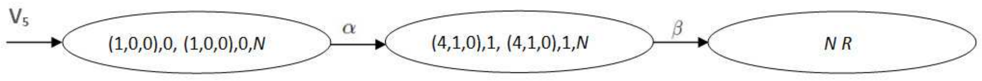


Figure 23: Verifier automaton for S_5^o

We further note that NR state is reachable for the automaton (V_5) in Fig. 23. Hence, the conditions in 7 are violated.

CHAPTER 4

MODULAR FAULT-RECOVERY SUPERVISORY CONTROL FOR FAULT RECOVERY

Fault-tolerant and fault-recovery control allow a system to continue its operation after a fault occurrence while fulfilling a potentially degraded specification. Generally, two types of fault-tolerance are considered [41]:

- **Passive fault tolerance:** Same supervisor is used as a controller for both the normal and faulty case.
- **Active fault-tolerance:** Needs a supervisor to control the system in case of fault occurrence. Such controller depends on a fault detection unit so as to adjust its operation in case of fault.

In this chapter, we propose computational procedures for modular fault-recovery supervisors. To this end, we employ and further develop methods that were first introduced for the supervisory control of reconfigurable manufacturing systems [42, 43]. The applicability of our method is demonstrated by a laboratory system at Çankaya University as shown in Fig. 24.

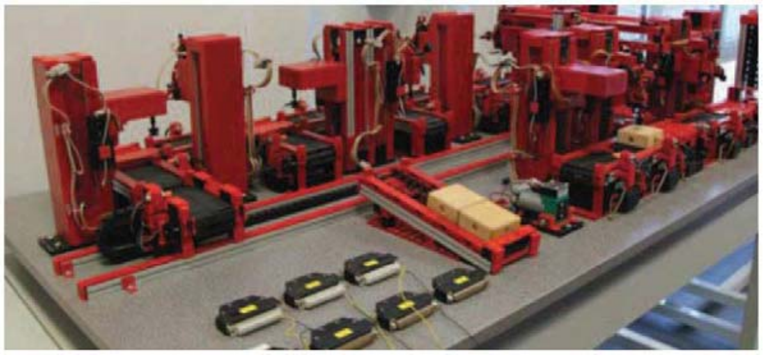


Figure 24: Laboratory manufacturing system

4.1. Monolithic Control for Fault Recovery

In this section, we outline an approach for the monolithic fault-recovery supervisory control [34]. This approach is the basis for our modular approach to fault-recovery supervisory control

Fault recovery systems are built based on three types of language specifications, a *nominal specification* which shows the acceptable behavior of the system before a fault occurrence, the *degraded specification* which is followed after fault occurrence before changing the fault recovery system over to the faulty specification, the *faulty specification* which should finally be followed in the faulty case.

4.1.1. Fault Recovery System Components

Fault recovery systems are always modeled by using the alphabets $\Sigma, \Sigma^{\mathrm{N}}, \Sigma^{\mathrm{F}}, \Sigma_{\mathrm{u}}$. Here, Σ^{F} represents fault events, Σ^{N} represents nominal events which have no relation with faults and $\Sigma = \Sigma^{\mathrm{N}} \dot{\cup} \Sigma^{\mathrm{F}}$. The set of uncontrollable events is Σ_{u}. The plant model in fault recovery systems are formalized using two different models, one model for the nominal(non faulty) plant behavior $G^{\mathrm{N}} = (X^{\mathrm{N}}, \Sigma, \delta^{\mathrm{N}}, x_0^{\mathrm{N}}, X_{\mathrm{m}}^{\mathrm{N}})$, and another model for the whole plant $G = (X, \Sigma, \delta, x_0, X_{\mathrm{m}})$ which contains the faulty plant behavior. The acceptable behavior of the system is specified as:

$$L(G^{\mathrm{N}}) \subseteq L(G) \text{ and } L_{\mathrm{m}}(G^{\mathrm{N}}) \subseteq L_{\mathrm{m}}(G) \tag{4.1}$$

The language specifications that fault recovery systems are built on, are Denoted as follows:

1. (K^{N}) is subset of the marked language of the whole plant ($K^{N} \subseteq L_{\mathrm{m}}(G)$), which demonstrates the nominal behavior of the system incase of fault free(the fault is forbidden) which is obtained by $SupC(K^{\mathrm{N}}, L(G^{\mathrm{N}}), \Sigma_{\mathrm{u}})$.
2. (K^{D}) is subset of the marked language of the whole plant ($K^{\mathrm{D}} \subseteq L_{\mathrm{m}}(G)$), which demonstrates the degraded specification(admissible behavior) which is the stage that the system follows after fault occurrence.
3. Finally, we show (K^{F}) is subset of the degraded specification (K^{D}), which demonstrates the faulty specification (wanted behavior), which is the stage that is reached after the fault occurrence in fault recovery systems.

4.1.2. Fault Recovery Supervisor Design Requirements

We introduce the faulty supervisor as $S^{\mathrm{F}} = (Q^{\mathrm{F}}, \Sigma, \nu^{\mathrm{F}}, q_0^{\mathrm{F}}, Q_{\mathrm{m}}^{\mathrm{F}})$. S^{F} is the supervisor that achieves fault recovery after following the three stages that we introduced before. Our aim now is to design a non blocking fault-recovery supervisor $S^{\mathrm{F}} = (Q^{\mathrm{F}}, \Sigma, \nu^{\mathrm{F}}, q_0^{\mathrm{F}}, Q_{\mathrm{m}}^{\mathrm{F}})$ for G and Σ_{u} such that:

$$(A) L_{\mathrm{m}}(G||S^{\mathrm{F}}) \cap (\Sigma^{\mathrm{N}})^{\star} \subseteq K^{\mathrm{N}} \tag{4.2}$$

(B) It holds for all $s \in L(G||S^{\mathrm{F}}) \cap (\Sigma^{\mathrm{N}})^{\star}\Sigma^{\mathrm{F}}(\Sigma^{\mathrm{N}})^{\star}$ that $s = s_1^1 s_1^2 \cdots s_k^1 s_k^2 \mathtt{f} s_3$ with $\mathtt{f} \in \Sigma^{\mathrm{F}}, s_i^j \in (\Sigma^{\mathrm{N}})^{\star}$ for $i = 1, \ldots, k$ and $j = 1, 2, s_1^1 \cdots s_k^1 \in \overline{K^{\mathrm{N}}}$ and $s_1^2 \cdots s_k^2 s_3 \in K^{\mathrm{D}}$ (4.3)

$$(C) K^{\mathrm{F}} \Leftarrow L_{\mathrm{m}}(G||S^{\mathrm{F}}) / \overline{K^{\mathrm{N}}}\Sigma^{\mathrm{F}} \tag{4.4}$$

In words, the items mean that the supervisor must agree with the nominal specification K^{N} if there is no fault. Second, the supervisor must follow the degraded specification K^{D} to continue system operation after any fault occurrence in the system. Hereby, a part of the substring ($s_1^1 \cdots s_k^1 \in \overline{K^{\mathrm{N}}}$) should fulfill nominal behavior before a fault occurrence in the system, whereas the other substring ($s_1^2 \cdots s_k^2 s_3 \in K^{\mathrm{D}}$) should continue to a string in K^{D} before the fault occurrence in the system. Hence, we say that the degraded specification (K^{D}) is completed using the normal behavior substring that fulfills (K^{N}). Last but not least, fault recovery supervisor must finally converge to the faulty behavior (K^{F}).

4.1.3. Supervisor for Fault Recovery and Repair

We introduce a new language specification ($K^{\mathrm{A}} \subseteq L_{\mathrm{m}}(G)$) as in [34]. This specification K^{A} fulfills the two conditions (A) and (B) in subsection (4.1.2). We note that:

$$\overline{K^{\mathrm{N}}}\Sigma_{\mathrm{F}}\Sigma^{\star} \cap L(G) \tag{4.5}$$

Follows the normal behavior of the system before any fault occurrence. Moreover, interleaving composition operation as introduced in Section 2.6 is used to obtain:

$$K^{\mathrm{A}} = (\overline{K^{\mathrm{N}}}\Sigma^{\mathrm{F}} ||| K^{\mathrm{D}}) \cap (\overline{K^{\mathrm{N}}}\Sigma^{\mathrm{F}}(\Sigma^{\mathrm{N}})^{\star} \cap L_{\mathrm{m}}(G)) \tag{4.6}$$

The new specification (K^{A}) as introduced above fulfills the two conditions (A and B). Then according to equation (4.6), the specification (K^{A}) follows all the strings in the plant such that a sub string of these strings until the occurrence of the fault will be in $\overline{K^{\mathrm{N}}}\Sigma^{\mathrm{F}}$, and the other sub string should continue to a string in K^{D} before the fault occurrence in the system. Conversely, the strings of K^{A} will exist in $\overline{K^{\mathrm{N}}}$ before a fault occurrence in the system. Then, the recovery specification is found by getting the prefix of the nominal behavior (K^{N}) which is done via marking (K^{N}) strings.

Now, after obtaining (K^{A}) , the conditions (A and B) are achieved by obtaining the supervisor($S^{\mathrm{A}} = (Q^{\mathrm{A}}, \Sigma, \nu^{\mathrm{A}}, q_0^{\mathrm{A}}, Q_{\mathrm{M}}^{\mathrm{A}})$) that realizes the maximally permissive behavior such that ($G||S^{\mathrm{A}}$) follows the nominal behavior before the fault occurrence and follows the admissible behavior (degraded specification) to continue system operation after fault occurrence. The supervisor is computed as follows:

$$L_{\mathrm{m}}(S^{\mathrm{A}}) = SupC(K^{\mathrm{A}}, L(G), \Sigma_{\mathrm{u}}). \tag{4.7}$$

Finally, condition (C) is achieved by using *language convergence* method which is introduced in Section 2.7. Fault recovery supervisor S^{F} is computed using the algorithm in [34] such that:

$$L_{\mathrm{m}}(S^{\mathrm{F}}) = \mathtt{ConvAfter}(K^{\mathrm{F}}, L_{\mathrm{m}}(S^{\mathrm{A}}), \overline{K^{\mathrm{N}}}\Sigma^{\mathrm{F}}, \Sigma_{\mathrm{u}}) \tag{4.8}$$

The function `ConvAfter` is implemented in `libfaudes`.

4.2. Modular Control Overview

Fault recovery supervisors are constructed to govern the systems that are subjected to faults using the same idea as supervisors for the reconfiguration control of DES. That is, it is desired to follow a certain behavior (configuration) until a change of configuration is requested. In that case, the current configuration is completed and the new configuration is started [42, 43]. The main difference is that the changing between the supervisors in reconfiguration control occur according to system requirements to perform a specific work. In contrast, in fault recovery supervisors there is no system demand to change the supervisor, rather system demand happens involuntarily due to a fault. In addition, it has to be noted that the plant behavior after fault is different since certain operations cannot be performed.

4.2.1. Problem Description

In order to simplify the notation, we consider a system that consists of two modular components (module 1 and module 2). Moreover, we assume that a fault can happen in module 1, whereas module 2 is fault-free but its operation is affected by the fault in module 1. In this setting, we use the fault-recovery controllers in Fig. 25.

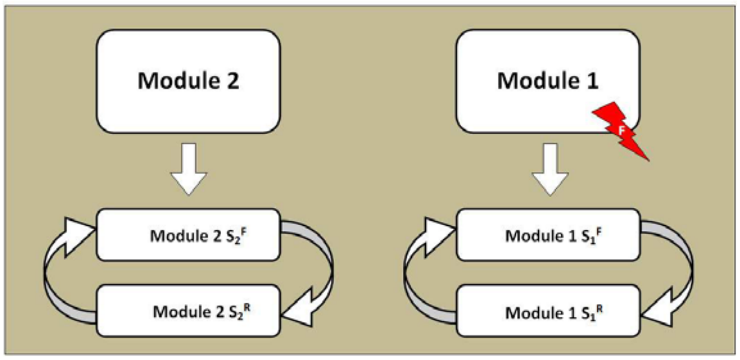


Figure 25: Whole system modules control

Each module is controlled by two supervisors. In module 1, S_1^{F} is active in the nominal case and during the transition to the faulty behavior after a fault occurrence. Reversely, S_1^{R} controls module 1 in the faulty case and returns module 1 to the nominal operation after repair. The supervisors S_2^{F} and S_2^{R} have an analogous function for module 2. In the following we develop design procedures for the previously described supervisors.

4.2.2. Example System

In order to illustrate the basic idea of fault-recovery, we introduce the components of a manufacturing system that we used in our examples in this chapter. The system overview is shown in Fig. 26.

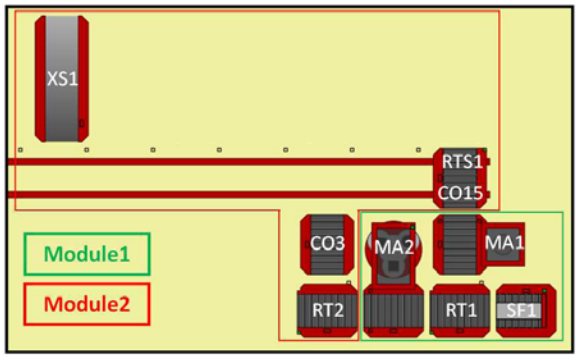


Figure 26: The entire example system

The system consists of the following components:

- One Stack Feeder (SF1).
- Two Rotary Tables (RT1 and RT2).
- Two Machine Tools (MA1 and MA2).
- One Rail Transport System (RTS1).
- One conveyor belt (CO3).
- One Exit Slide (XS1).

We next give a description of the components.

Stack Feeder (SF1):

The stack feeder is a device that works as product collector and system feeder. It pushes the products to the system when it is allowed. The products that enter the system through the stack feeder, move to the next component which is the component beside the stack feeder (RT1 in our system).

The figure and the model of SF1 are shown in Fig. 27, and the events of the stack feeder are listed in Stack feeder information table.

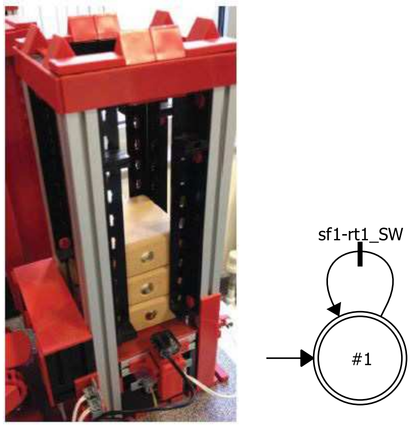


Figure 27: Stack feeder component and its model G_{SF1}

Table 1: Stack feeder G_{SF1} model information

Event names	**Illustration**	**Type**
`sf1-rt1_SW`	The product moves from sf1 to rt1	Controllable

Rotary Table (RT1):

Rotary table is equipped with a belt in order to transport products to different directions as shown in Fig. 28. The rotary table can rotate either with clockwise direction and stay vertically to transport the product UP or Down, or it can rotate with anticlockwise direction and stay horizontally to transport the product LEFT or Right. In our example SF1 is to the right, MA2 is to the left and MA1 is above RT1. Hence, the rotary table can transport and receive products from and to these devices.

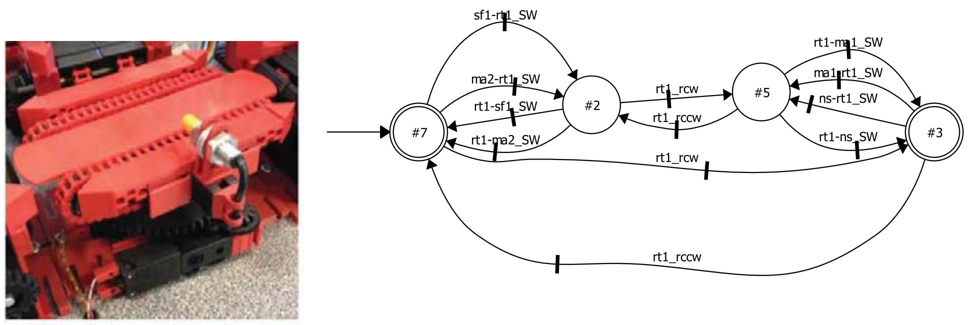


Figure 28: Rotary table component and its model G_{RT1}

Table 2: Rotary table G_{RT1} model information

Event names	Illustration	Type
`sf1-rt1_SW`	The product moves from sf1 to rt1	Controllable
`rt1-sf1_SW`	The product moves from rt1 to sf1	Controllable
`ma2-rt1_SW`	The product moves from ma2 to rt1	Controllable
`rt1-ma2_SW`	The product moves from rt1 to ma2	Controllable
`ma1-rt1_SW`	The product moves from ma1 to rt1	Controllable
`rt1-ma1_SW`	The product moves from rt1 to ma1	Controllable
`ns-rt1_SW`	The product moves from ns to rt1	Controllable
`rt1-ns_SW`	The product moves from rt1 to ns	Controllable
`rt1_rcw`	The rotary table1 rotates (clockwise)	Controllable
`rt1_rccw`	The rotary table1 rotates (anticlockwise)	Controllable

Machine (MA1):

The machine as in Fig. 29 consists of several parts: a moving belt which transports products, the machine head that moves up or down in order to enable processing, a single machine tool which performs a production operation. In our system, MA1 is located between RT1 and a conveyor belt CO15.

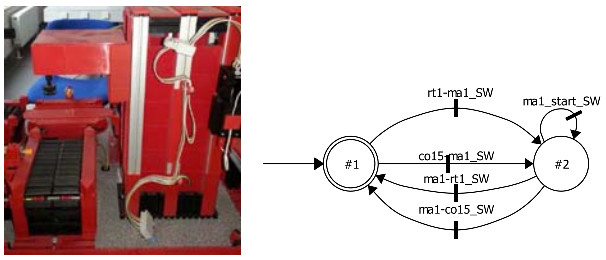


Figure 29: Manufacturing machine component and its normal model G_{MA1}

In our example, we assume that the fault can happen in MA1 such that MA1 will not be able to start processing any more. The model of MA1 with fault-occurrence is shown in Fig. 30.

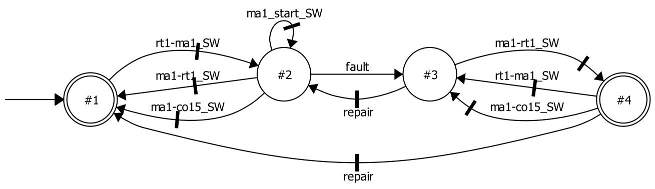


Figure 30: Manufacturing machine model with fault G^{F}_{ma1}

Table 3: Manufacturing machine G_{MA1} model information

Event names	**Illustration**	**Type**
`rt1-ma1_SW`	The product moves from rt1 to ma1	Controllable
`co15-ma1_SW`	The product moves from co15 to ma1	Controllable
`ma1-rt1_SW`	The product moves from rt1 to ma2	Controllable
`ma1-co15_SW`	The product moves from ma1 to co15	Controllable
`fault`	Fault happens in the system	Uncontrollable
`repair`	Repair happens in the system	Controllable
`ma1_start_SW`	Start processing in (ma1)	Controllable

Machine (MA2):

MA2 follows the same operation as MA1 with the different neighbors RT1 (right) and RT2 (left). The model of MA2 is shown in Fig. 31.

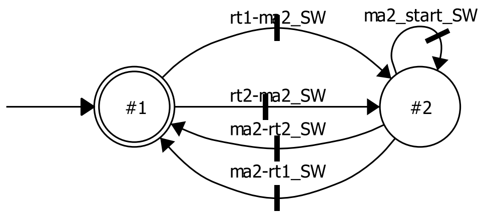


Figure 31: Manufacturing machine model G_{ma2}

Table 4: Manufacturing machine G_{MA2} model information

Event names	**Illustration**	**Type**
`rt1-ma2_SW`	The product moves from rt1 to ma2	Controllable
`rt2-ma2_SW`	The product moves from rt2 to ma2	Controllable
`ma2-rt1_SW`	The product moves from ma2 to rt1	Controllable
`ma2-rt2_SW`	The product moves from ma2 to rt2	Controllable
`ma2_start_SW`	Start processing in (ma2)	Controllable

Exit Slides (XS1 and XS2):

Exit slide allows storage of products after production and is shown in Fig. 32. It simply receives products from the neighboring component. The models of XS1 and XS2 are shown in Fig. 32.

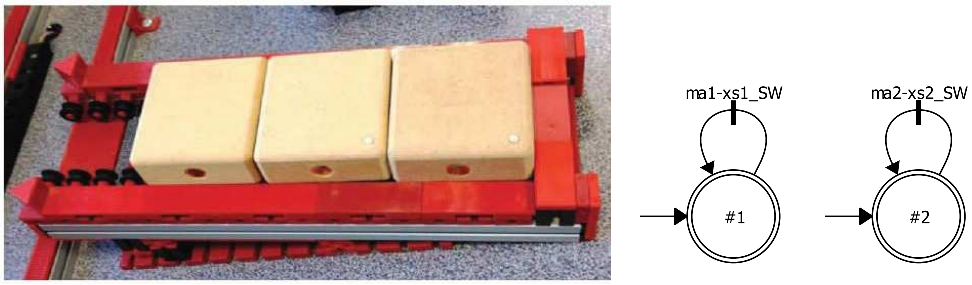


Figure 32: Exit Slide component and its models G_{XS1}, G_{XS2}

Table 5: Exit slides G_{XS1}, G_{XS2} models information

Event names	**Illustration**	**Type**
`ma1-xs1_SW`	The product moves from ma1 to xs1	Controllable
`ma2-xs2_SW`	The product moves from ma2 to xs2	Controllable

Rail Transport System (RTS):

The rail transport system (RTS) consists of a *rail* that allows two cars to move left and right, and a *conveyor belt* on each car that can transport products as is shown in Fig. 33. In this thesis ,we only consider the right car denoted as RTS1. The rail is divided into 4 positions (5,4,3,2) and the RTS1 can move to each position.

Each position of the rail transport system has an upper and a lower component such that the conveyor belt (CO) can transport products up or down. The conveyor belts on our RTS is denoted as CO15.

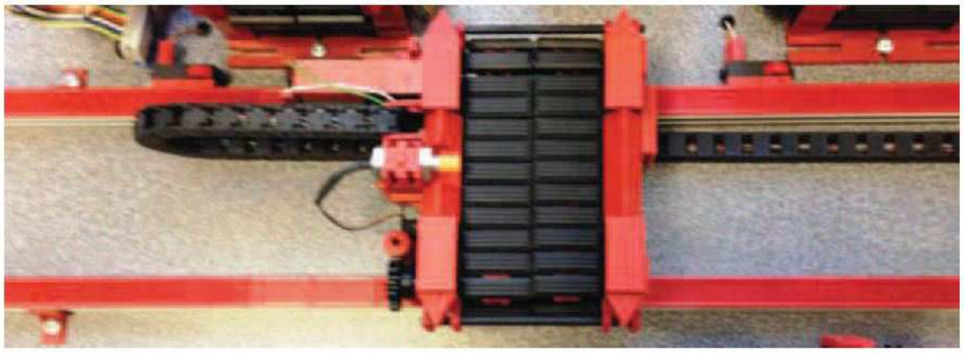


Figure 33: Rail transport system RTS component

Plant models of RTS1 and CO15 are shown in Fig. 34 and 35, respectively. Hereby, it is assumed that RTS1 is initially in position 5 and CO15 is empty.

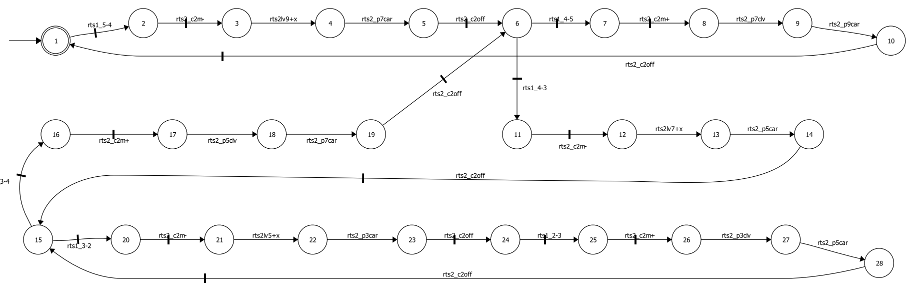


Figure 34: Rail transport system plant model G_{RTS}

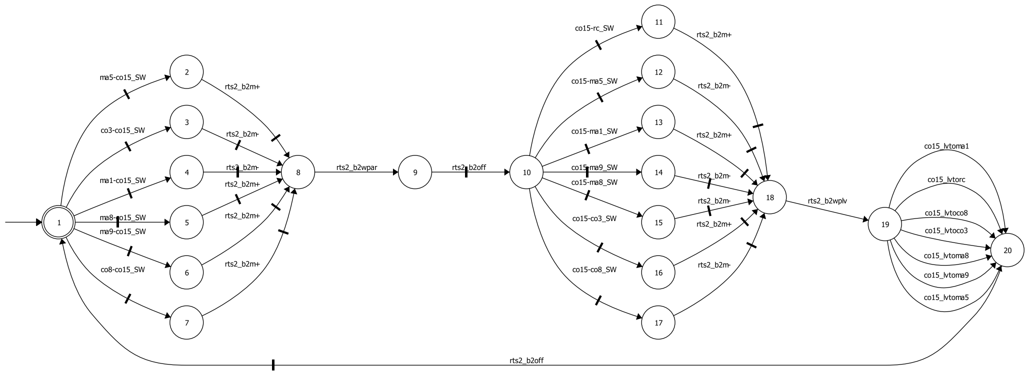


Figure 35: Conveyor belt plant model G_{CO15}

The plant model for both CO15 and RTS1 is built by using synchronous composition operation of the following components:

$$G_{RTS1} = G_{CO15} || G_{RTS} \tag{4.9}$$

Next, it is required to determine specifications such that each position only allow product transport to and from the correct components as in the real system. These specifications are shown in the following figures.

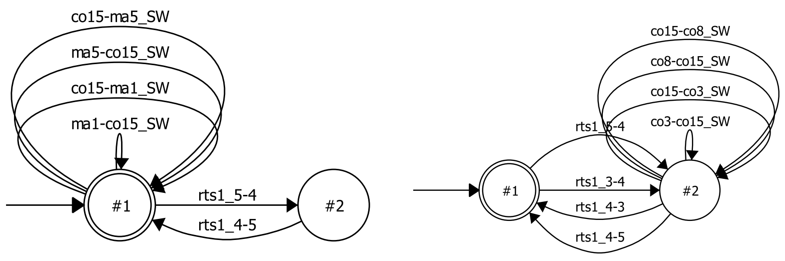


Figure 36: Position 5 specification C_{P5} and position 4 specification C_{P4}

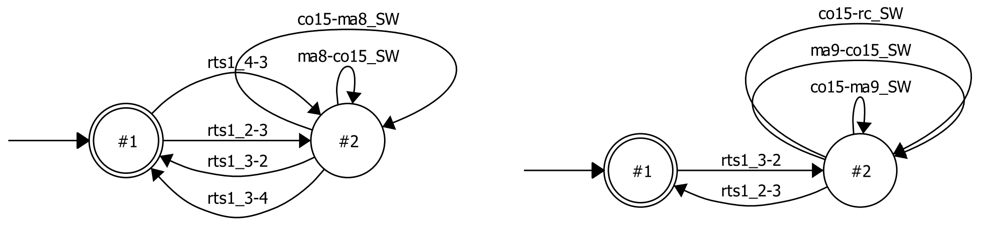


Figure 37: Position 3 specification C_{P3} and position 2 specification C_{P2}

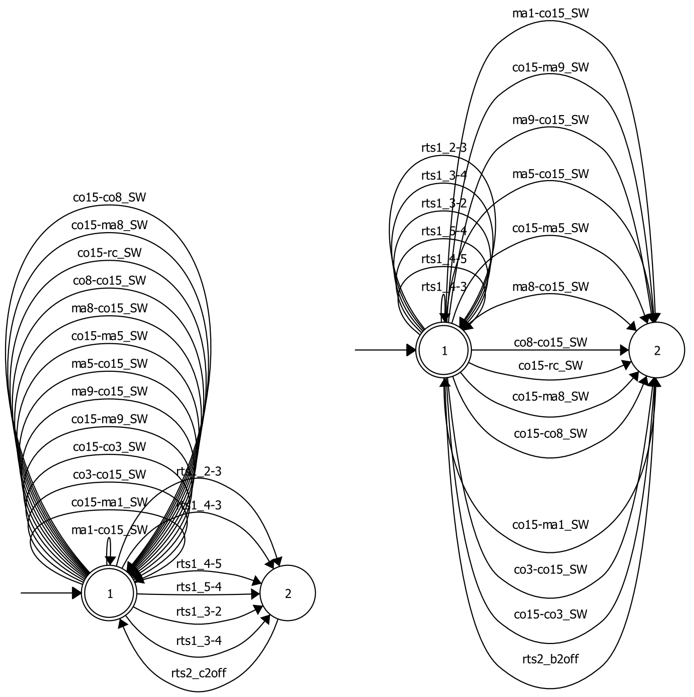


Figure 38: Mutual exclusion specifications C_{1orig} and C_{2orig}

Then, the overall specification is computed using synchronous composition operation of the specifications above:

$$C_{Positions} = C_{P5}||C_{P4}||C_{P3}||C_{P2}||C_{1orig}||C_{2orig} \tag{4.10}$$

As a result, RTS1 allows product transport to and from the correct components of the laboratory system.

The result of the closed-loop RTS G^{Low}_{RTS1} that comprises both RTS1 and CO15 is computed using the SupCon Algorithm:

$$L_{\text{m}}(G^{\text{Low}}_{RTS1}) = SupC(G^{\text{Low}}_{RTS1}, L_{\text{m}}(C_{Position}), \Sigma_{\text{u}}) \tag{4.11}$$

Since the overall result is too big for display in this thesis, we only note that G^{Low}_{RTS1} has **89 states**. In our design, we use an abstraction of G^{Low}_{RTS1} according to [44] to get a small plant automata with less number of states (G_{RTS1}). The result is shown in Fig. 39.

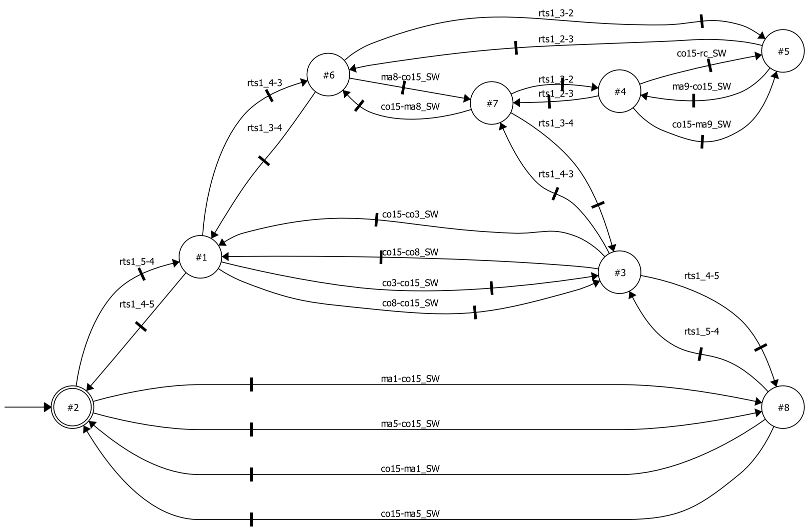


Figure 39: The model of RTS1 G_{RTS1}

Table 6: Rail transport system G_{RTS1} model information

Event names	Illustration	Type
`rts1_4-5`	Move co15 from position4 to position5	Controllable
`co15-ma1_SW`	The product moves from co15 to ma1	Controllable
`ma1-co15_SW`	The product moves from ma1 to co15	Controllable
`co15-ma5_SW`	The product moves from co15 to ma5	Controllable
`ma5-co15_SW`	The product moves from ma5 to co15	Controllable
`rts1_5-4`	Move co15 from position5 to position4	Controllable
`co15-co3_SW`	The product moves from co15 to co3	Controllable
`co3-co15_SW`	The product moves from co3 to co15	Controllable
`co15-co8_SW`	The product moves from co15 to co8	Controllable
`co8-co15_SW`	The product moves from co8 to co15	Controllable
`rts1_4-3`	Move co15 from position4 to position3	Controllable
`co15-ma8_SW`	The product moves from co15 to ma8	Controllable
`ma8-co15_SW`	The product moves from ma8 to co15	Controllable
`rts1_3-4`	Move co15 from position3 to position4	Controllable
`rts1_3-2`	Move co15 from position3 to position2	Controllable
`co15-rc_SW`	The product moves from co15 to rc	Controllable
`co15-ma9_SW`	The product moves from co15 to ma9	Controllable
`ma9-co15_SW`	The product moves from ma9 to co15	Controllable
`rts1_2-3`	Move co15 from position2 to position3	Controllable

Conveyor Belt (CO3):

CO3 moves products to two directions (Up,Down), see Fig. 40. In our laboratory system, CO3 transports products from and to RT2 and CO15. The model of CO3 is shown in Fig. 40.

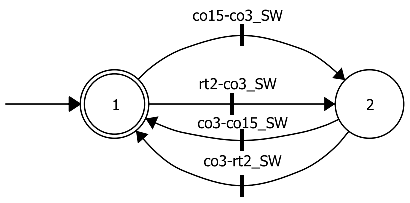


Figure 40: Conveyor belt component and its model G_{CO3}

Table 7: Conveyor belt G_{CO3} model information

Event names	**Illustration**	**Type**
`co15-co3_SW`	The product moves from co15 to co3	Controllable
`co3-co15_SW`	The product moves from co3 to co15	Controllable
`rt2-co3_SW`	The product moves from rt2 to co3	Controllable
`co3-rt2_SW`	The product moves from co3 to rt2	Controllable

Rotary Table (RT2):

The operation of RT2 is analogous to the previously described RT1. Only the neighbor components CO3 and MA2 are different. The model of RT2 is shown in Fig. 42.

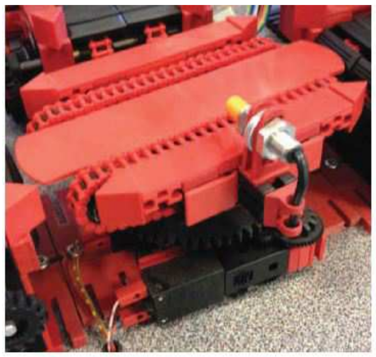


Figure 41: Rotary table component

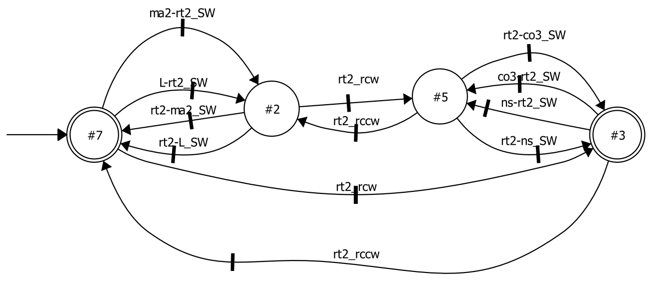


Figure 42: Rotary table model G_{RT2}

Table 8: Rotary table G_{RT2} model information

Event names	**Illustration**	**Type**
`ma2-rt2_SW`	The product moves from ma2 to rt2	Controllable
`rt2-ma2_SW`	The product moves from rt2 to ma2	Controllable
`L-rt2_SW`	The product moves from left side to rt2	Controllable
`rt2-L_SW`	The product moves from rt2 to left side	Controllable
`co3-rt2_SW`	The product moves from co3 to rt2	Controllable
`rt2-co3_SW`	The product moves from rt2 to co3	Controllable
`ns-rt2_SW`	The product moves from ns to rt2	Controllable
`rt2-ns_SW`	The product moves from rt2 to ns	Controllable
`rt2_rcw`	The rotary table2 rotates (clockwise)	Controllable
`rt2_rccw`	The rotary table2 rotates (anticlockwise)	Controllable

4.2.3. Definition of Modules

We finally refer again to Fig. 26. We divide this system into two modules according to Section 4.2.1. Module 1 consists of SF1, RT1, MA1 and MA2 and hence is the module with the faulty component MA1. Module 2 consists of RT2, CO3 and RTS1. There is no fault in module 2 but it might be affected by a fault in module 1.

4.3. Computation of Modular Supervisors for Fault Recovery

In this thesis, each module is controlled using four supervisors that are synchronized to govern the faulty and the non-faulty system behavior as shown in Fig. 26. We next develop algorithms for computing these supervisors in order to achieve fault recovery and return to the nominal behavior after system repair.

4.3.1. Fault-Recovery for the Faulty Module

We introduce a new algorithm called `SystemFaultN1` to build the supervisor S_1^{F} that governs module 1 in case of a fault.

Algorithm 1. *(`SystemFaultN1`):*

$$\textit{Compute } S^F = (Q^F, \Sigma^F, \nu^F, q_0^F, Q_m^F) \textit{ using } G, K^N, K^D, K^F, \Sigma_u \textit{ as in } [34] \tag{4.12}$$

$$\textit{Find } q \in Q^F \textit{ such that } L_m(S_q^F) \subseteq K^F \tag{4.13}$$

$$\forall \sigma \in \Sigma^F \textit{ such that } \nu^F(q,\sigma)! : \textit{ remove the transition from } q \textit{ with } \sigma \textit{ from } \nu^F \tag{4.14}$$

$$Q^F = Q^F \cup \{wait\} \tag{4.15}$$

$$Q_m^F = Q_m^F \cup \{wait\} \tag{4.16}$$

$$\nu^F(q, repair\_fin) := wait \tag{4.17}$$

$$\forall \sigma \in \Sigma^F \setminus \{repair\_st\} : \nu^F(wait,\sigma) = wait \tag{4.18}$$

$$\forall \sigma \in \Sigma^F \setminus \{repair\_st\} : \nu^F(wait,\sigma) = wait \tag{4.19}$$

$$\nu^F(wait, repair\_st) = q_0^F \tag{4.20}$$

$$\textit{Compute } Acc(S^F) \textit{ and return the result} \tag{4.21}$$

In words, we first obtain S^{F} using the algorithm in [34]. Then, we find the state q such that the marked language of S^{F} starting from q is contained in the marked language of K^{F}. This means, we look for the state where the desired faulty behavior is achieved. We keep this state but remove all transitions from that state. Instead we insert a new transition with event $repair\_fin$ to a new waiting state $wait$. All events except for $repair\_st$ are selflooped in $wait$ and $repair\_st$ leads to the initial state q_0^{F}. Finally, make the result accessible. That is, the resulting supervisor is responsible for leading module 1 to the faulty behavior. Here, the event $repair\_fin$ indicates that the faulty behavior is reached. After achieving this, S_1^{F} becomes inactive in the state $wait$ and becomes active again if repair is completed with the event $repair\_st$. The basic operation of S_1^{F} is shown in Fig. 43.

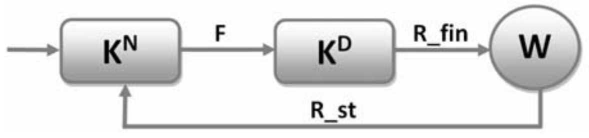


Figure 43: System fault supervisor diagram

First Module Example:

We consider module 1 as shown in Fig. 44 with SF1, MA1, MA2 and RT1. In the nominal case, we want to enter products from SF1 and produce in MA1. Then, products should exit the system to XS1 (blue arrow). If a fault happens in MA1, products that are already transported to MA1 should go back to MA2 for processing and exit from XS2 (green arrow). New products should directly move to MA2 (red arrow).

Note: We changed the name of XS1 to CO15 and XS2 to RT2 intentionally in the supervisors for compatibility issues during operating our laboratory example.

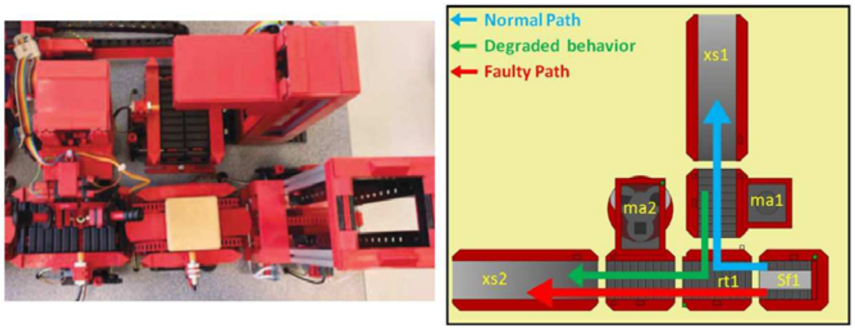


Figure 44: Module 1 components

In order to design a fault recovery supervisor S_1^{F}, we use the plant model of module 1:

$$G_1 = G_{SF1} || G_{RT1} || G_{MA1}^{\mathrm{F}} || G_{MA2} \tag{4.22}$$

Next, we need to obtain the nominal specification K_1^{N} by specifying a closed loop system that allows the tasks presented in the nominal behavior. K_1^{N} is computed from the automata in Fig. 45 as:

$$K_1^{\mathrm{N}} = L_{\mathrm{m}}(C_1^{\mathrm{N1}}) || L_{\mathrm{m}}(C_2^{\mathrm{N1}}) \tag{4.23}$$

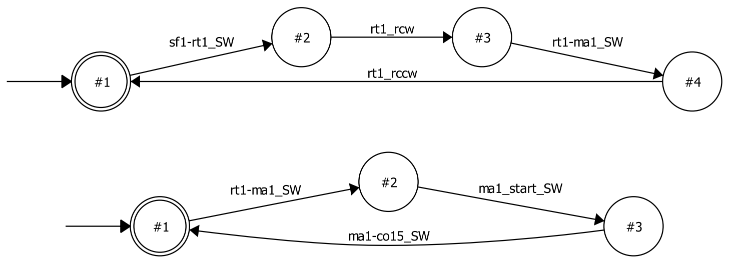


Figure 45: Nominal specification automata C_1^{N1} and C_2^{N1}

Now, we formulate the degraded specification K_1^{D} composed of $C_1^{\mathrm{D}}, C_2^{\mathrm{D}}, C_3^{\mathrm{D}}$ in Fig. 46 as:

$$K_1^{\mathrm{D}} = L_{\mathrm{m}}(C_1^{\mathrm{D1}})||L_{\mathrm{m}}(C_2^{\mathrm{D1}})||(L_{\mathrm{m}}(C_3^{\mathrm{D1}}) \tag{4.24}$$

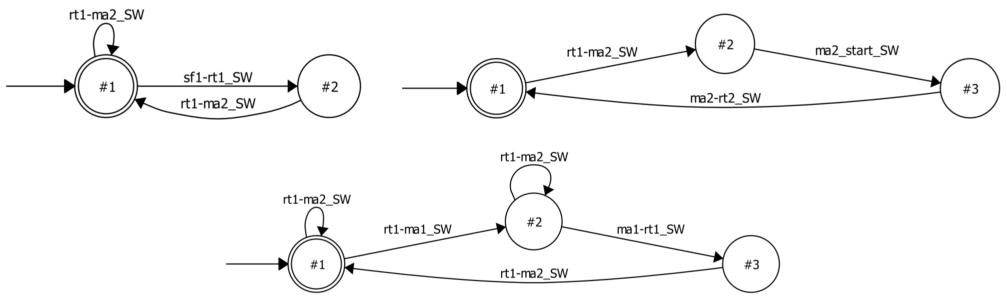


Figure 46: Degraded specification automata C_1^{D1}, C_2^{D1}, C_3^{D1}

We note that processing with machine MA1 is not possible according to K_1^{D} and C_2^{D1} shows that MA2 is used in case of fault.

Finally, we use the faulty specification K_1^{F} with the automata in Fig. 47.

$$K_1^{\mathrm{F}} = L_{\mathrm{m}}(C_1^{\mathrm{F1}})||L_{\mathrm{m}}(C_2^{\mathrm{F1}}) \tag{4.25}$$

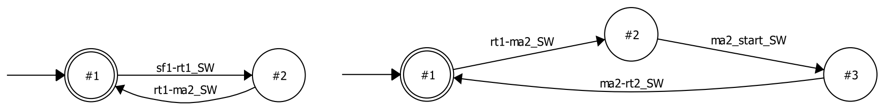


Figure 47: Faulty specification automata C_1^{F1} and C_2^{F1}

The specifications $C_1^{\mathrm{F}}, C_2^{\mathrm{F}}$ show that new products can enter the system through SF1, move to MA2 and leave the system after processing.

Applying Algorithm 1, we obtain the supervisor S_1^{F} with 30 states as shown in Fig. 48. Note that this figure is only included in the thesis to highlight the structure of S_1^{F}. The upper part of the supervisor implements the nominal behavior, whereas the center part realizes the degraded behavior. The wait state (with selfloops) is finally reached when the desired faulty behavior is achieved.

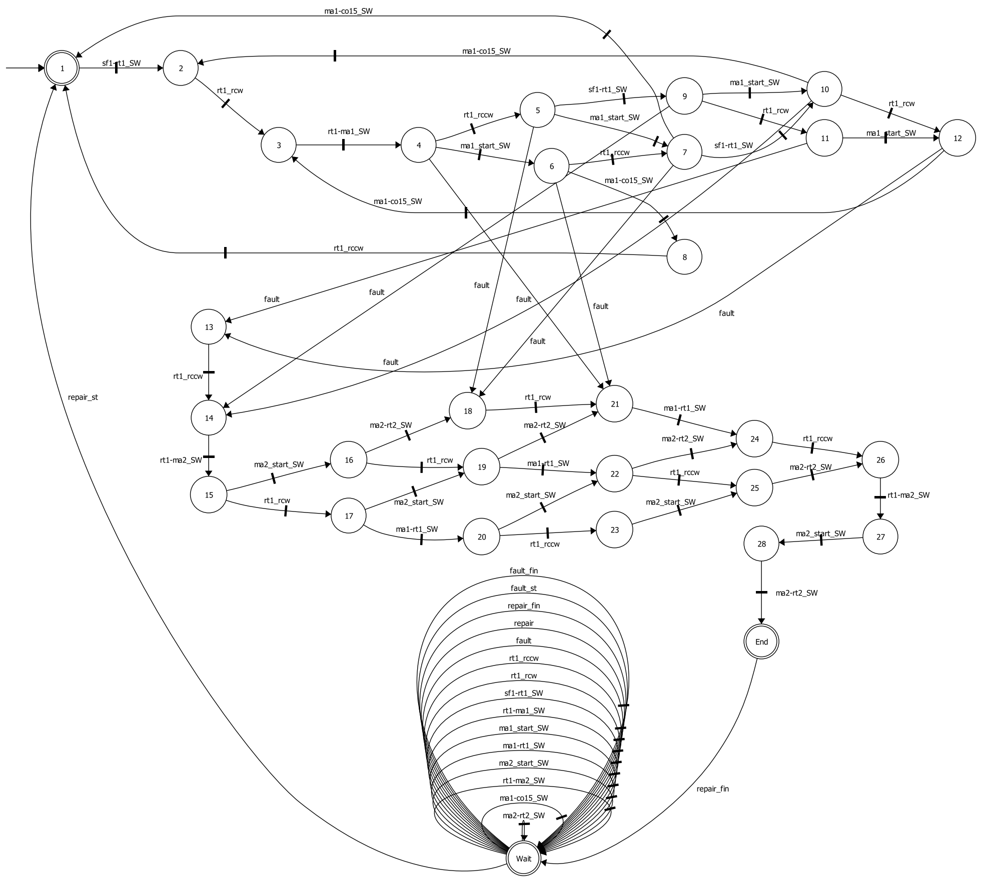


Figure 48: Fault-recovery supervisor S_1^{F}

4.3.2. Fault-Recovery for Non-Faulty Module

The supervisor that controls the non-faulty module 2 in case of fault occurrence is computed according to a new algorithm `SystemFaultN2`. The algorithm computes the supervisor S_2^{F} exactly as `SystemFaultN1` in Section 4.3.1 with the small modification that step 3 is removed from Algorithm 1. This measure is taken under the assumption that module 2 should expect further products that come from module 1 when performing the degraded behavior. The completion of the degraded behavior is decided by module 1 when entering the faulty behavior with the event $repair\_fin$.

Second Module Example:

Module 2 comprises different components as shown in Fig. 49. These components are RTS1, RT2, CO3 and XS2.

Note: We changed XS2 name intentionally to MA9 in the supervisors for compatibility issues during operation of our laboratory model.

Fig. 49 shows the desired system paths. The blue arrow represents the normal path, the green arrow represents the path that the system should follow after the fault.

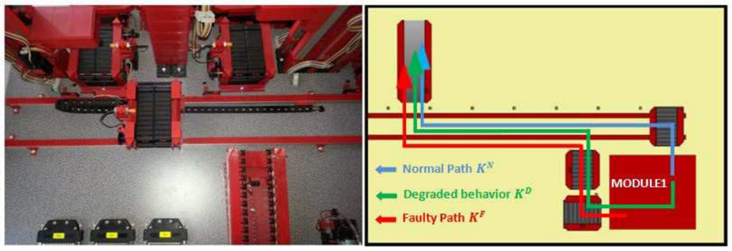


Figure 49: Module 2 components

Similar to module 1, we compute the supervisor S_2^{F} according to the modified Algorithm 1. First, we need the plant G_2 for module 2 that is computed by using synchronous composition operation:

$$G_2 = G_{RTS1}||G_{CO3}||G_{RT2} \tag{4.26}$$

The resulting automaton has **128 states** which is too big to be shown in our thesis.

Next, we obtain the nominal specification K_2^{N} that the system follows in module 2 in case of no fault in the system. We use the automata in Fig. 50 to compute:

$$K_2^{\mathrm{N}} = L_{\mathrm{m}}(C_1^{\mathrm{N2}})||L_{\mathrm{m}}(C_2^{\mathrm{N2}})||L_{\mathrm{m}}(C_3^{\mathrm{N2}} \tag{4.27}$$

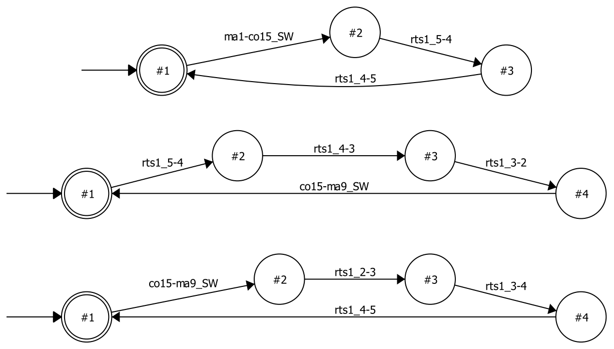


Figure 50: Nominal specification automata $C_1^{\mathrm{N2}}, C_2^{\mathrm{N2}}, C_3^{\mathrm{N2}}$

The specifications show that the product leaves module 1 and enters module 2 through CO15 in position 5 of RTS1. Then, it moves to the positions 4,3,2. In position 2, the product moves to the exit slide MA9 and then RTS1 returns to position 5.

Next, we determine the degraded specification K_2^{D} for module 2 to continue the product path after a fault. Then, the specifications that perform this task is specified by the automata in Fig. 51 as:

$$K_2^{\mathrm{D}} = C_1^{\mathrm{D2}}||C_2^{\mathrm{D2}}||C_3^{\mathrm{D2}}||C_4^{\mathrm{D2}}||C_5^{\mathrm{D2}}||C_6^{\mathrm{D2}} \tag{4.28}$$

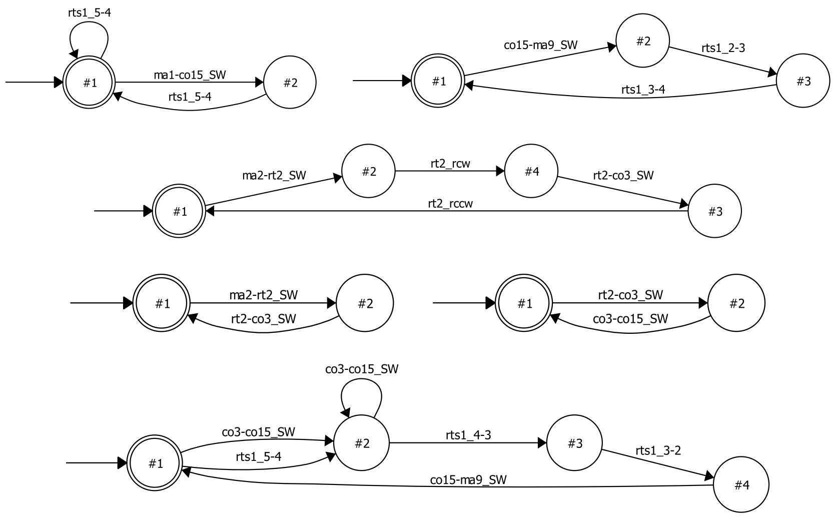


Figure 51: Degraded specification automata $C_1^{\mathrm{D2}}, C_2^{\mathrm{D2}}, C_3^{\mathrm{D2}}, C_4^{\mathrm{D2}}, C_5^{\mathrm{D2}}, C_6^{\mathrm{D2}}$

After the fault occurrences in the system, module 2, RTS1 moves from position 5 to position 4 and waits for products to enter from RT2. When a product reaches RT2, it turns clockwise, then moves the product to CO3. In order to allow a new product to enter module 2, RT2 is turned anticlockwise and CO3 moves the product to the waiting RTS1 in position 4. Next, RTS1 moves to position 4,3,2 and the product leaves the system to MA9. Finally, the empty RTS2 moves back to position 4 to wait a new product from the faulty path of the system.

Finally, we specify the faulty behavior of module 2 based on the automata in Fig. 52 as:

$$K_2^{\mathrm{F}} = C_1^{\mathrm{F2}}||C_2^{\mathrm{F2}}||C_3^{\mathrm{F2}}||C_4^{\mathrm{F2}}||C_5^{\mathrm{F2}} \tag{4.29}$$

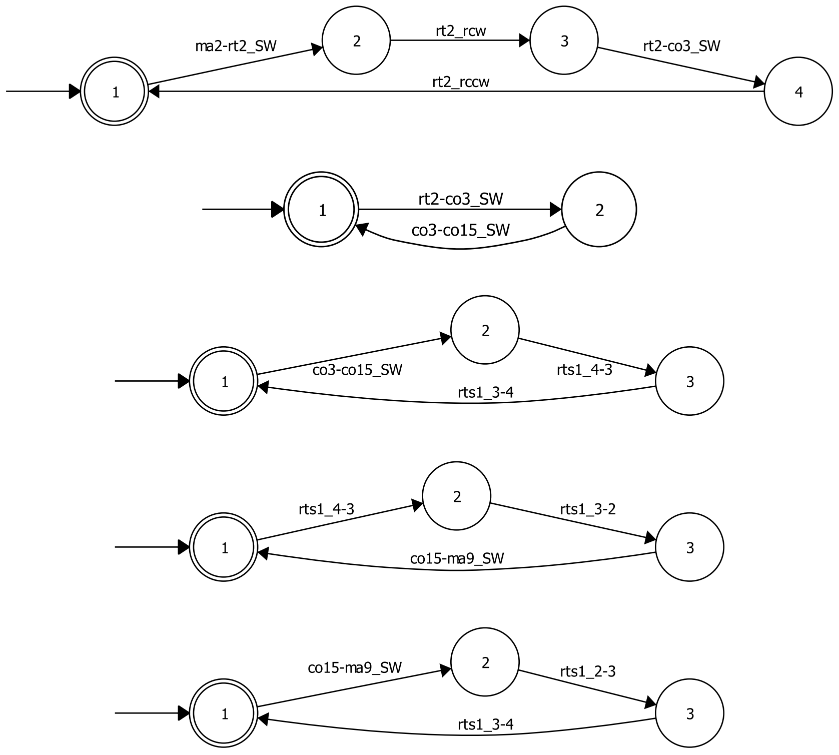


Figure 52: Faulty specification automata $C_1^{\mathrm{F2}}, C_2^{\mathrm{F2}}, C_3^{\mathrm{F2}}, C_4^{\mathrm{F2}}, C_5^{\mathrm{F2}}$

In the faulty behavior of module 2, the product enters module 2, RT2 turns clockwise, then moves the product to CO3. In order to allow a new product to enter module 2, RT2 turns anticlockwise, after that CO3 moves the product to the waiting RTS1 in position

4. Next, CO15 moves over the positions (4,3,2), then the product leaves the system to MA9. Finally, the empty RTS1 returns to position 4.

Using the plant and specifications for module 2, we apply the algorithm `SystemFaultN2` and obtain the supervisor S_2^{F} for module 2 with **115 states** which is too big to be put in our thesis.

4.3.3. Repair for the Faulty Module

We next design system repair supervisor denoted as $S^{\mathrm{R}} = (Z^{\mathrm{R}}, \Sigma^{\mathrm{R}}, \alpha^{\mathrm{R}}, z_0^{\mathrm{R}}, Z_{\mathrm{m}}^{\mathrm{R}})$. We develop the algorithm `SystemRepair`. Note that this algorithm will be suitable for computing the supervisors under repair S_1^{R} and S_2^{R} for both modules. As inputs, we use the following automata:

plant $G = (X, \Sigma, \delta, x_0, X_{\mathrm{m}})$ (4.30)

Supervisor $S = (Q, \Sigma, \nu, q_0, Q_{\mathrm{m}})$ that realizes the faulty system behavior. (4.31)

Attractor supervisor $T_1 = (Q, \Sigma, \omega, q_0, -\ )$ for state attraction of the set $\{q_0\}$ in S. (4.32)

Attractor supervisor $T_2 = (Y, \Sigma, \lambda, y_0, Y_{\mathrm{m}})$ for state attraction of the set $\{x_0\}$ in G. (4.33)

State $\hat{x} \in X$ that corresponds to state q_0 in S. (4.34)

Algorithm 2. *(`SystemRepair`):*

$$Z^R = Q \cup \{q' | q \in Q\} \cup \{wait\} \cup \{x' | x \in X\} \tag{4.35}$$

$$Z_m^R = Q_m \cup \{wait\} \tag{4.36}$$

$$z_0^R = \{wait\} \tag{4.37}$$

$$\begin{aligned} &\textit{For each } q \in Q \textit{ and } \sigma \in \Sigma: \\ &\nu(q,\sigma)! \Rightarrow \alpha^R(q,\sigma) = \nu(q,\sigma). \\ &\omega(q,\sigma)! \Rightarrow \alpha^R(q',\sigma) = \tilde{q}' \textit{ for } \omega(q,\sigma) = \tilde{q} \end{aligned} \tag{4.38}$$

$$\begin{aligned} &\textit{For each } q \in Q: \\ &\alpha^R(q, repair) = q' \end{aligned} \tag{4.39}$$

$$\begin{aligned}&\text{For each } \sigma \in \Sigma^R \setminus \{fault\_st\}:\\ &\alpha^R(wait,\sigma) = wait\end{aligned} \tag{4.40}$$

$$\begin{aligned}&\text{For each } \sigma \in \Sigma^R \setminus \{fault\_st\}:\\ &\alpha^R(wait,\sigma) = wait\end{aligned} \tag{4.41}$$

$$\alpha^R(wait, fault\_st) = q_0 \text{ and } \alpha^R(q_0', fault\_fin) = \hat{x} \tag{4.42}$$

$$\text{Set } \hat{x} \text{ as initial state in } T_2 \tag{4.43}$$

$$\text{Set } y_0 \in Y \text{ as marked state in } T_2 \tag{4.44}$$

$$\text{Trim } T_2 \tag{4.45}$$

$$\text{Copy } T_2 \text{ in } S^R \tag{4.46}$$

$$\alpha^R(q_0', fault\_fin) = \hat{x} \tag{4.47}$$

$$\begin{aligned}&\text{For } y \in Y \text{ and } \sigma \in \Sigma \text{ such that } \lambda(y,\sigma) = y_0:\\ &\alpha^R(q,\sigma) = wait(q \text{ is the state of } S^R \text{ that corresponds to } y)\end{aligned} \tag{4.48}$$

The basic structure of S^R is also shown in Fig. 53.

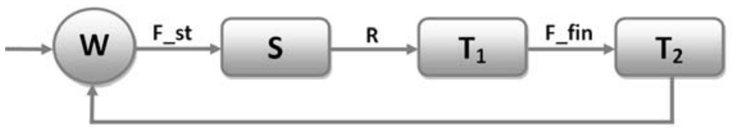


Figure 53: System repair supervisor diagram

In words, S^R obtains the states of S, T_1, T_2 and the waiting state *wait* and the marked states of S and *wait*. The initial state of S^R is *wait* (initially the supervisors S_1^F and S_2^F are active). The transitions of the supervisor S, T_1 and T_2 are directly copied into S^R.

In addition, a transition with the event *repair* is inserted from each state q of S to the corresponding state q' of T_1. The initial state of T_1 is connected to $\hat{x}$ via $fault\_fin$ and the marked state of T_2 is connected to the waiting state via the transitions that are leading to the initial state of G in the attractor T_2.

In our example, we use the supervisor under fault S as shown in Fig. 54. It is computed from the plant automaton G_1 and the faulty specification K_1^F.

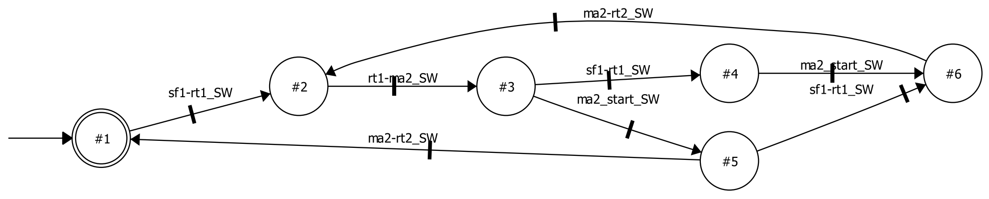


Figure 54: Supervisor S_1 for module 1

The automaton T_1 is obtained from S by removing the transitions from state 1 to 2 and from state 6 to 2. Moreover, T_2 consists of a single state without any transitions. Then, we can apply Algorithm 2 to find the system repair supervisor S_1^{R}. The result is shown in Fig. 55.

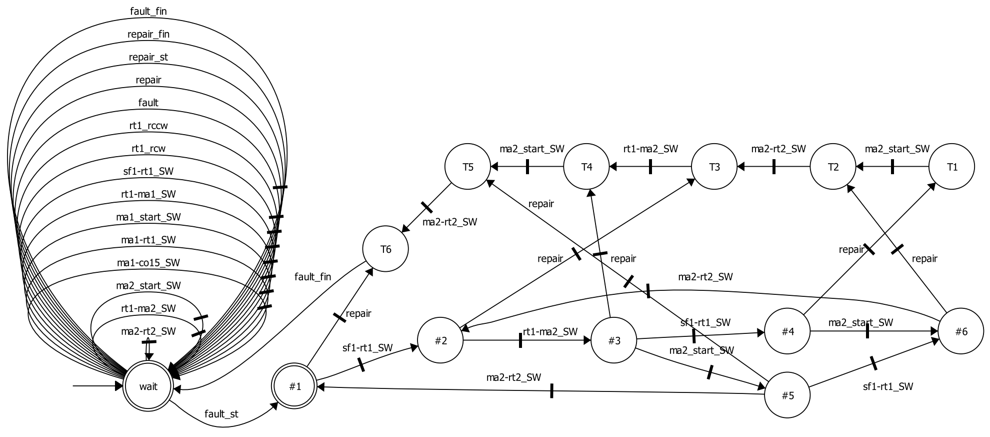


Figure 55: System repair supervisor S_1^{R}

System repair supervisor (S_1^{R}) in Fig. 55 shows the three parts S_1, T_1, wait as discussed before.

4.3.4. Repair for Non-Faulty Module

The construction of the repair supervisor for module 2 follows the same Algorithm 2 (`SystemRepair`). Hence, we just demonstrate the application of this algorithm to our example. Again, the supervisor S is obtained from G_2 and the fault specification K_2^{F}. Note that this computation assumes that RTS1 starts from position 4 and the initial state of G_2 is chosen accordingly. Since S and T_1 have 48 states, they cannot be shown in this thesis. In addition, it holds that the operation of T_1 terminated in position 4 of RTS1, whereas the operation of the non-faulty system behavior should start from position 5. Hence, an automaton T_2 that moves RTS1 from position 4 to position 5 with the event $rts1\_4-5$ is chosen. Then, the application of Algorithm 2 leads to a system repair supervisor S_2^{R} with 98 states. Again, the automaton is too big to be displayed in this thesis.

4.3.5. Fault and Repair Coordinators

The fault-recovery supervisors constructed in the previous sections are responsible for leading the system to the faulty behavior in case of a fault occurrence and return to the nominal system behavior in case of system repair. What is missing is the coordination of when the faulty/nominal system behavior should start. To this end, we use two coordinator automata for each module as introduced in [43].

The first coordination automaton is denoted as P_1 and is responsible for starting the faulty system behavior after the degraded system behavior is completed. It is defined as follows:

$$\begin{aligned} P_1 = (\{first, second, last\}, \Sigma \cup \{fault, repair\_fin, fault\_st\},\\ \eta_1, first, \{first\}) \end{aligned} \tag{4.49}$$

$$\begin{aligned} &\text{For each } \sigma \in \Sigma \setminus \{fault\}: \\ &\eta_1(first, \sigma) = first \end{aligned} \tag{4.50}$$

$$\eta_1(first, fault) = second \tag{4.51}$$

$$\begin{aligned} &\text{For each } \sigma \in \Sigma \setminus \{repair\_fin\}: \\ &\eta_1(second, \sigma) = second \end{aligned} \tag{4.52}$$

$$\eta_1(second, repair\_fin) = last \tag{4.53}$$

$$\eta_1(last, fault\_st) = first \tag{4.54}$$

In words, P_1 has three states that are connected according to the event sequence $fault$, $repair\_fin$, $fault\_st$ such that the faulty system behavior is only allowed to start ($fault\_st$) if the degraded behavior is finished ($repair\_fin$). Selfloops are added in the states $first$ and $second$, whereas no further events are allowed in $last$. Hence, the faulty behavior must start immediately.

The second coordination automaton is denoted as P_2 and is responsible for starting the nominal system behavior after the behavior under repair is completed.

Its definition is analogous to P_1 with a small modification in state *last* depending on the attractor T_2, such that all events that appear in T_2 are added in the form of a selfloop transition in state *last* of P_2.

$$P_2 = (\{first, second, last\}, \Sigma \cup \{repair, fault\_fin, repair\_st\}, \eta_2, first, \{first\}) \tag{4.55}$$

$$\text{For each } \sigma \in \Sigma \setminus \{fault\} : \quad \eta_2(first, \sigma) = first \tag{4.56}$$

$$\eta_2(first, repair) = second \tag{4.57}$$

$$\text{For each } \sigma \in \Sigma \setminus \{fault\_fin\} : \quad \eta_2(second, \sigma) = second \tag{4.58}$$

$$\eta_2(second, fault\_fin) = last \tag{4.59}$$

$$\eta_2(last, repair\_st) = first \tag{4.60}$$

$$\text{For all events } \sigma \text{ of } T_2 : \quad \eta_2(last, \sigma) = last \tag{4.61}$$

We next determine the coordinators for module 1 and 2 of our example system. $P_{1,1}$ governs the start of the faulty behavior of module 1 and $P_{1,2}$ governs the start of the nominal behavior of module 1. The respective automata are shown in Fig. 56 and 57.

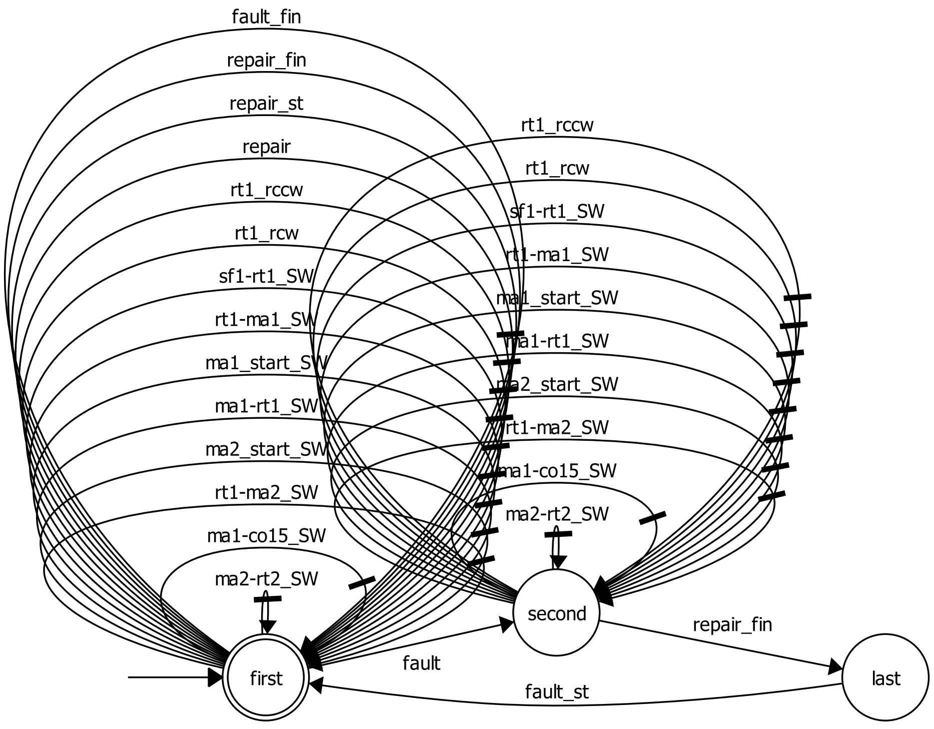


Figure 56: Coordinating automaton $P_{1,1}$ for S_1^{F}

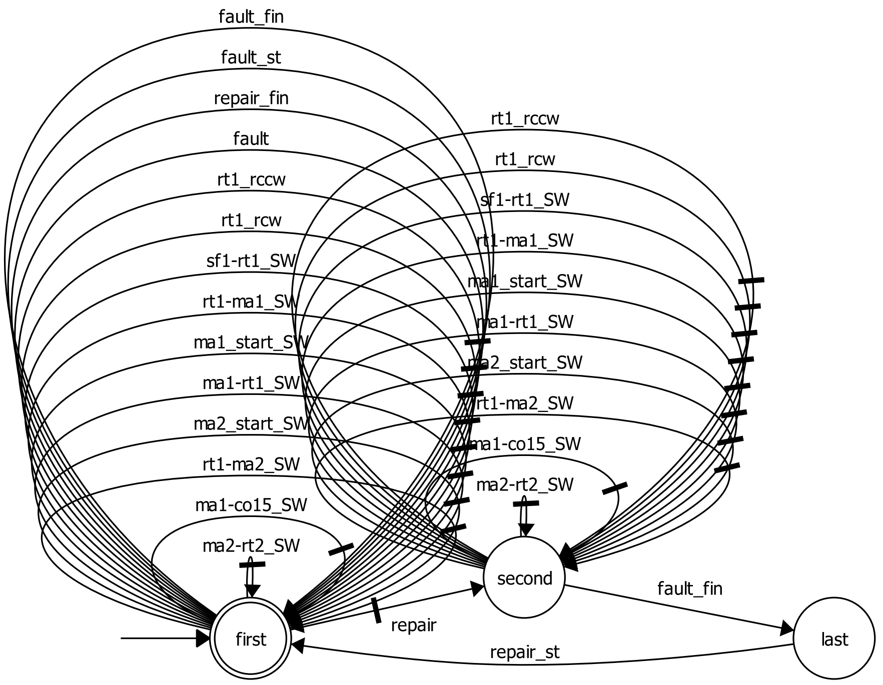


Figure 57: Coordinating automaton $P_{1,2}$ for S_1^{R}

It is clear in this example that no selfloops are added in the state *last* of $P_{1,2}$ because the attractor T_2 of the faulty module does not have transitions.

Similarly, we obtain the coordination automata $P_{2,1}$ and $P_{2,2}$ in Fig. 58 and 59.

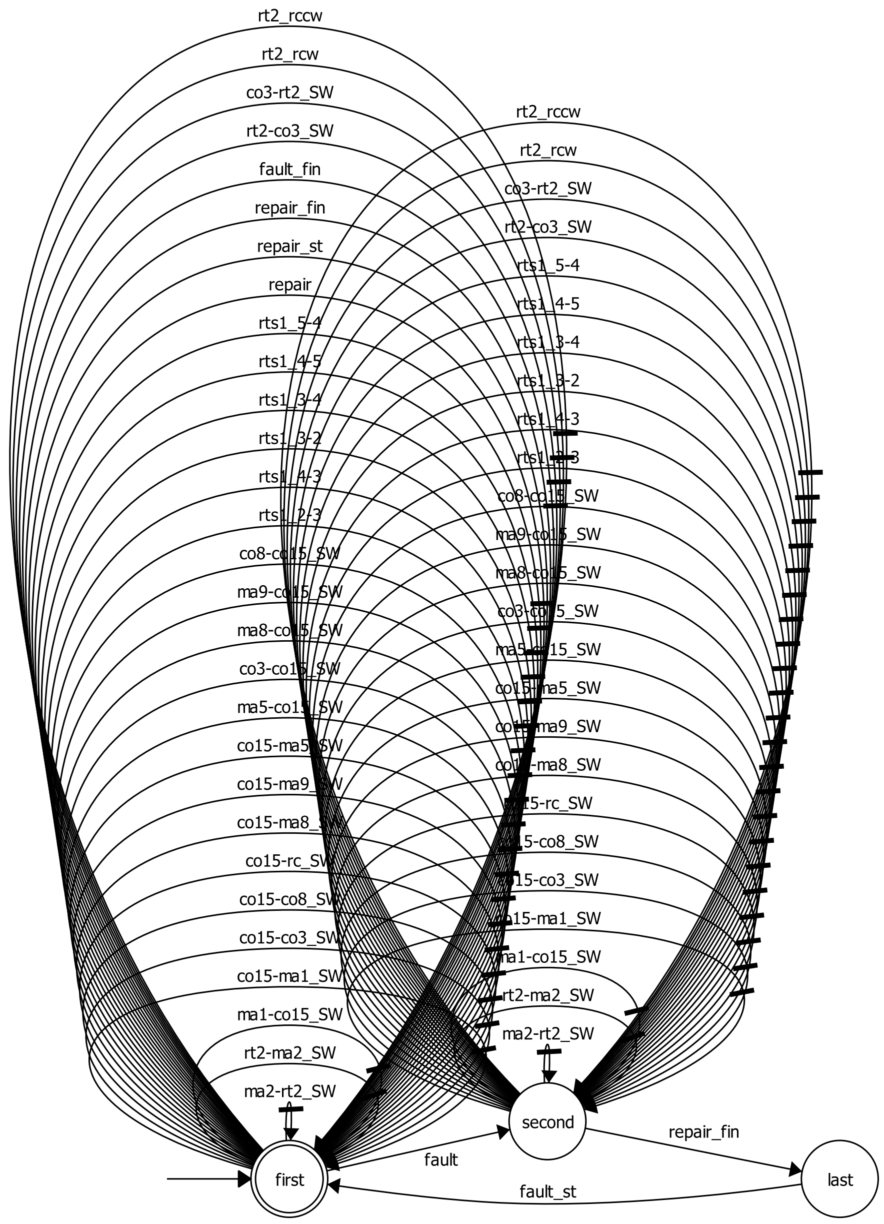


Figure 58: Coordinating automaton $P_{2,1}$ for S_2^{F}

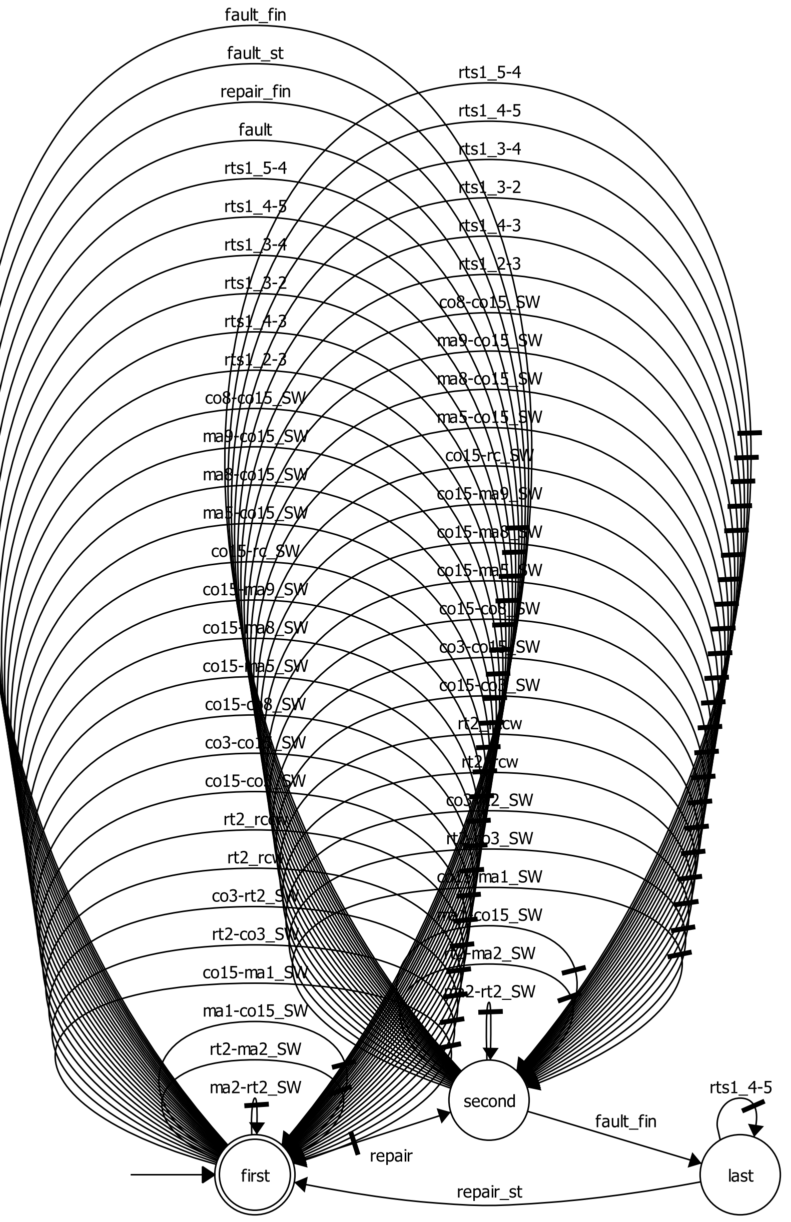


Figure 59: Coordinating automaton $P_{2,2}$ for S_2^{R}

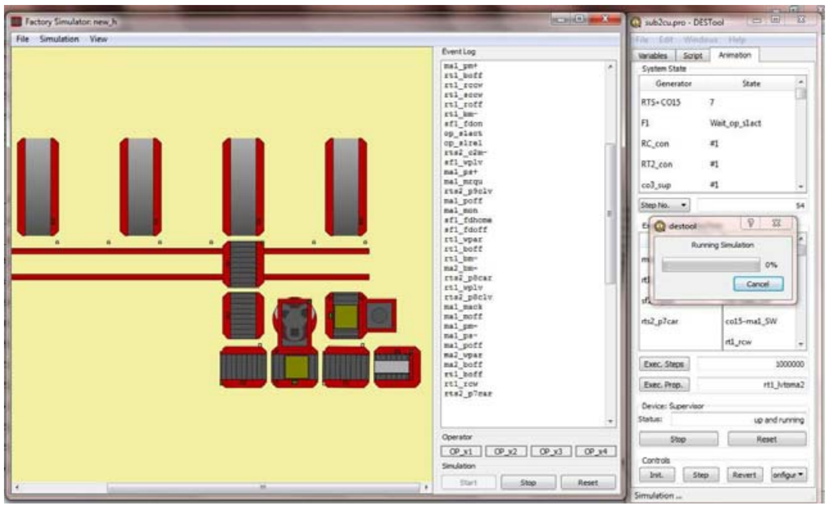
Here, we can see a selfloop with event $rts1\_4-5$ in *last* of $P_{2,2}$ since attractor T_2 of module 2 has two states and one event (`rts1_4-5`).

In summary, we use the modular supervisor as shown in the following equation in order to implement the overall supervisor.

$$S_1^{\mathrm{F}}||S_2^{\mathrm{F}}||S_1^{\mathrm{R}}||S_2^{\mathrm{R}}||P_{1,1}||P_{1,2}||P_{2,1}||P_{2,2} \tag{4.62}$$

4.4. Faulty And Non-Faulty Modules Simulation

In addition to the theoretical computation, it is possible to validate the designed supervisors using the manufacturing simulator `FlexFact` [45] and the controller simulator `DESTool` [46]. We used this simulator for the example system in Fig. 60. It could be verified that the closed-loop operation is as desired when using our modular failure-recovery supervisors. In addition, a laboratory experiment with the laboratory system in Fig. 24 was performed successfully.



```
ma1_pm+
rt1_boff
rt1_rccw
rt1_sccw
rt1_roff
rt1_bm-
sf1_fdon
op_slact
op_slrel
rts2_c2m-
sf1_wplv
ma1_ps+
ma1_mrqu
rts2_p9clv
ma1_poff
ma1_mon
sf1_fdhome
sf1_fdoff
rt1_wpar
rt1_boff
rt1_bm-
ma2_bm-
rts2_p8car
rt1_wplv
rts2_p8clv
ma1_mack
ma1_moff
ma1_pm-
ma1_ps-
ma1_poff
ma2_wpar
ma2_boff
rt1_boff
rt1_rcw
rts2_p7car
```

Generator	State
RTS+CO15	7
F1	Wait_op_slact
RC_con	#1
RT2_con	#1
co3_sup	#1

```
Step No.      54
rts2_p7car    co15-ma1_SW
              rt1_rcw
Exec. Steps   1000000
Exec. Prop.   rt1_lvtoma2
Status:       up and running
```

Figure 60: Whole system simulation using flexfact program

CHAPTER 5

CONCLUSION AND FUTURE WORK

5.1. Conclusion

We are gathering two different methods to create our idea in this thesis. We first use the idea of fault recovery of discrete event systems idea to compute supervisors that govern the system behavior under the fault based on the algorithm in [34]. Such fault recovery supervisor follows behaviors: the nominal behavior that realizes the desired system behavior without fault, the degraded behavior that continues the system behavior after fault with a reduced performance, the faulty behavior which realizes the desired behavior under fault and is achieved after a bounded delay. Secondly, we use the idea of reconfiguration supervisors for system repair. Specifically, the system behavior is returned to follow the nominal behavior. The combination of both ideas is found suitable in this thesis.

As the main contribution of the thesis, the cited methods are not used in a monolithic way but to design modular fault-recovery supervisors. To this end, we focus on the case of systems with two modular components, whereby one modular component can become faulty. We develop construction algorithms for separate modular supervisors that handle the system behavior after a fault and after system repair. Due to the modular design, it is possible to apply our method to large-scale system. It also has to be noted that this is the first modular approach to failure-recovery for discrete event systems in the existing literature. The practicability of the developed method is illustrated by a medium-size manufacturing system example and is validated by simulation and laboratory experiment.

5.2. Future Work

The thesis work considers a special case of modular control for failure-recovery. Accordingly, several extensions are possible:

- It is possible to consider more than two modular plant components.
- It is possible to consider faults in more than one modular plant component.
- In order to coordinate the behavior or the modular system components, it is possible to extend the modular approach by hierarchical supervisory control.
- Consider under which conditions recovery from a fault does not affect the rest of a large system.

REFERENCES

1. **C. G. Cassandras and S. Lafortune,(2008),**“*Introduction to Discrete Event Systems,SecondEdition,*” Springer.

2. **S.R. Das and L.E. Holloway,(2000),** “*Characterizing A Confidence Space for Discrete event Timings for Fault Monitoring Using Discrete Sensing and Actuation Signals*,”Systems, Man and Cybernetics, Part A: Systems and Humans, IEEE Transactions on, vol 30, no. 1, pp. 52–66.

3. **Shengbing Jiang, Zhongdong Huang, V. Chandra, and R. Kumar, (2001),**“*A Polynomial Algorithm for Testing Diagnosability of Discrete Event Systems,*” Automatic Control, IEEE Transactions on, vol 46, no. 8, pp. 1318–1321.

4. **A. Bouloutas, G.W. Hart, and M. Schwartz,(1992),** “Simple *Finite-State Fault Detectors for Communication Networks,*” Communications, IEEE Transactions on,vol. 40, no. 3, pp. 477–479.

5. **A. Benveniste, E. Fabre, S. Haar, and C. Jard,(2003),**“*Diagnosis of Asynchronous Discrete Event Systems: A Net Unfolding Approach,*” Automatic Control, IEEETransactions on, vol. 48, no. 5, pp. 714–727, 2003.

6. **D.N. Godbole, J. Lygeros, E. Singh, A. Deshpande, and A.E. Lindsey, (2000),**“*Communication Protocols for A Fault-Tolerant Automated Highway System,*” Control Systems Technology, IEEE Transactions on, vol 8, no. 5, pp. 787–800.

7. **P. J. RamadgeandW. M.Wonham, (1987),** “*Supervisory Control of A Class of Discrete Event Processes,*” SIAM J. Control Optim., vol. 25, no. 1, pp. 206–230.

8. **M. Sampath, R. Sengupta, S. Lafortune, K. Sinnamohideen, and D. Teneketzis, (1995),**“*Diagnosability of Discrete-Event Systems,*” Automatic Control, IEEE Transactions on, vol. 40, no. 9, pp. 1555–1575.

9. **S. Yoo and S. Lafortune, (2002),**“*Polynomial Time Verification of Diagnosability of Partially Observed Discrete Event Systems*,” Automatic Control, vol. 47, no. 9, pp. 1491–1495.

10. **A. Schumann and Y. Pencol´e, (2007),**“*Scalable Diagnosability Checking of Event Driven System*,” In International Joint Conference on Artificial Intelligence, Hyderabad, India, pp. 575–580.

11. **T. Yoo and H. Garcia, (Dec 2008),**“*Diagnosis of Behaviors of Interest in Partially Observed Discrete Event Systems*,” System and Control Letters, vol. 57, no. 12, pp. 1023–1029.

12. **K. W. Schmidt, (2010),**“*Abstraction-Based Failure Diagnosis for Discrete Event Systems*,”System and Control Letters, vol. 59, pp. 42–47.

13. **R. Debouk, S. Lafortune, and D. Teneketzis, (2000),**“*Coordinated Decentralized Protocols for Failure Diagnosis of Discrete Event Systems*,” Journal of Discrete Event Dynamic Systems: Theory and Applications, vol. 10, pp. 33–86.

14. **Y. Pencol´e and M. Cordier, (2005),** “*A Formal Framework for The Decent Ralised Diagnosis of Large Scale Discrete Event Systems and its Application to Telecommunication Networks*,” Artif. Intell. Journal, vol. 164, no. (1-2), pp. 121–170.

15. **W. Qiu and R. Kumar, (March 2006),**“*Decentralized Failure Diagnosis of Discrete Event Systems*,” Systems, Man and Cybernetics, Part A: Systems and Humans, IEEE Transactions on, vol. 36, no. 2, pp. 384–395.

16. **R. Debouk, R. Malik, and B. Brandin,(2002),** “*A Modular Architecture for Diagnosis of Discrete Event Systems*,” In IEEE Conference on Decision and Control, Las Vegas, Nevada USA.

17. **O. Contant, S. Lafortune, and D. Teneketzis,(2006),**“*Diagnosabilityof Discrete Event Systems with Modular Structure*,” Discrete Event Dynamic Systems:Theory and Applications, vol. 16, pp. 9–17.

18. **C. Zhou, R. Kumar, and R.S. Sreenivas, (May, 2008),**“*Decentralized Modular Diagnosis of Concurrent Discrete Event Systems*,” In Discrete Event Systems, International Workshop on, Göteborg, Sweden, pages 388–393.

19. **W. Qiu and R. Kumar, (May 2008),**“*Distributed Diagnosis Under Bounded-Delay Communication of Immediately Forwarded Local Observations*,” Systems, Man and Cybernetics, Part A: Systems and Humans, IEEE Transactions on, vol. 38, no. 3, pp. 628–643.

20. **Shengbing Jiang, Ratnesh Kumar, and Humberto E. Garcia,(2003),**“*Diagnosis of Repeated/Intermittent Failures in Discrete Event Systems*,” Robotics and Automation, IEEE Transactions on, vol 19, no. 2, pp. 310–323.

21. **Shengbing Jiang and Ratnesh Kumar,(2006),** “*Failure Diagnosis of Discrete Event Systems with Linear-Time Temporal Logic Fault Specifications*,” Automation Science and Engineering, IEEE Transactions on, vol 3, no. 1, pp. 128–133.

22. **C. Zhou and R. Kumar, (2009),**“*Computation of Diagnosable Fault-Occurrence Indices for Systems with Repeatable Faults*,” IEEE Trans. Automat. Contr., vol. 54, no. 7, pp. 1477–1489.

23. **Hu Hong, (2012),** “*Diagnosis of Intermittent Faults in Discrete Event Systems*,” Master’s thesis, Department of Electrical and Computer Engineering, Toronto University, Toronto, Canada.

24. **K. W. Schmidt,(2011–2014),** “*A Formal Framework and Continuous Workflow for The Controller Design*,” Failure Diagnosis and Failure Recovery of Reconfigurable Manufacturing Systems, Career Project, TÜBİTAK.

25. **LibFAUDES, (2006–2014),** “*Libfaudes Software Library for Discrete Event Systems*,”[Online]. Available: www.rt.eei.uni-erlangen.de/FGdes/faudes(Data Download Date: 16.06.2013).

26. **A. Saboori and S. Hashtrudi-Zad, (2005)**, “*Fault Recovery in Discrete Event Systems*,” In Proc. Computational Intelligence: Methods and Applications, ICSC Congress on, Istanbul, Turkey.

27. **A. Paoli and S. Lafortune, (2005),** “*Safe Diagnosability for Fault Tolerant Supervision of Discrete Event Systems*,” Automatica, vol. 41, no. 8, pp. 1335–1347.

28. **A. Paoli, M. Sartini, and St´ephaneLafortune, (April 2011),** "*Active Fault Tolerant Control of Discrete Event Systems Using Online Diagnostics*," Automatica, vol. 47, no. 4, pp. 639–649.

29. **R. Kumar and S. Takai,(2012),** "*A Framework for Control-Reconfiguration Following Fault-Detection in Discrete Event Systems*," International Symposium on Fault Detection, Supervision and Safety of Technical Processes, pp. 848–853.

30. **T. Wittmann, J. Richter, and T. Moor, (2012),** "*Fault-Tolerant Control of Discrete Event Systems Based on Fault-Accommodating Models*," In 8th IFAC Symposium on Fault Detection, Supervision and Safety of Technical Processes, pages 854–859.

31. **Q. Wen, R. Kumar, J. Huang, and H. Liu, (2008),**"*A Framework for Fault-Tolerant Control of Discrete Event Systems*," Automatic Control, IEEE Transactions on, vol. 53, no. 8, pp. 1839–1849.

32. **Q. Wen, R. Kumar, and J. Huang, (2008),** "*Synthesis of Optimal Fault-Tolerant Supervisor for Discrete Event Systems*," In American Control Conference, pages 1172 –1177.

33. **A. Sülek and K. W. Schmidt, (2013),**"*Computation of Fault-Tolerant Supervisors for Discrete Event Systems*," In 4th IFAC Workshop on Dependable Control of Discrete Systems, pages 115–120.

34. **A. Sülek and K. W. Schmidt, (2014),** "Computation of Supervisors for Fault-Recovery and Repair for Discrete Event Systems," In Workshop on Discrete Event Systems.

35. **Y. Brave and M. Heymann, (1990),**"*Stabilization of Discrete Event Processes*," Int.J. Control, vol. 51, pp. 1101–1117.

36. **Y. Brave and M. Heymann, (1993),**"*On Optimal Attraction of Discrete Event Processes*,"Information Sciences, vol. 67, pp. 245–276.

37. **C. A. R. Hoare,(2004,1995),**"*Communicating Sequential Processes, Prentice Hall International*,".

38. **R. Kumar, V. Garg, and Steven I. Marcus, (1993),**"*Language Stability and Stabilizability of Discrete Event Dynamical Systems*," SIAM Journal of Control and Optimization, vol. 31, no. 5, pp. 1294–1320.

39. **Y.M. Willner and M. Heymann, (1995),**"Language Convergence in Controlled Discrete Event Systems," Automatic Control, IEEE Transactions on, vol. 40, no. 4, pp. 616 –627.

40. **K. W. Schmidt,(2013),** "*Fault Detection and Diagnosability of Discrete Event Systems with Recurring Faults*," Technical Report, Çankaya University, Ankara.

41. **M. Blanke, M. Kinnaert, J. Lunze, and M. Staroswiecki,(2010),**"*Diagnosis and fault-Tolerant Control*,"Springer.

42. **HarithM.KhalidHendi,(2014),** "*Applications of Reconfigurable Manufacturing Systems: A Laboratory Case Study*," Master's thesis, Department of Electronic and Communication Engineering, Çankaya University, Ankara, Turkey.

43. **Harith M. Khalid, Mustafa SancayKırık, and Klaus Werner Schmidt, (2013),**"Abstraction-Based Supervisory Control for Reconfigurable Manufacturing Systems," In Workshop on Dependable Control of Discrete Systems, York, United Kingdom, pp. 157–162.

44. **K.W. Schmidt, and C. Breindl, (2014),**"*A Framework for The Stabilization of Discrete Event Systems Under Partial Observation*," Information Sciences (in press).

45. **FlexFact, (2014),**"*FlexfactSimulator for Manufacturing Systems*,"[Online]. Available: http://www.rt.techfak.fau.de/FGdes/flexfact.html. (Data Download Date: 23.10.2013).

46. **DEStool, (2008–2014),**"*Destool Graphical User Interface for Discrete Event Systems*," [Online]. Available: http://www.rt.techfak.fau.de/FGdes/destool/. (Data Download Date: 02.04.2013).

APPENDICES A

CURRICCULUM VITAE

PERSONAL INFORMATION

Surname, Name: Mahmood, SarmadNozadMahmood

Nationality: Iraqi (IRAQ)

Date and Place of Birth: 27 October 1985, Kirkuk - Iraq

Marital Status: Single

Phone: +90 534 415 65 24 / +964 770 138 49 00

Email: sarmad.nozad@yahoo.com

EDUCATION

Degree	Institute	Year
MS.	Çankaya Univ. Electronic and Communication Eng.	2013/2014
B.Sc.	College of Technology, Kirkuk	2007/2008
High School	Al-Taakhi Preparatory School	2003/2004

FOREIGN LANGUAGES

Arabic, English, Turkish

PUPLICATIONS

Swash S. Muhammed, SarmadN. Mahmood, and Aydin. Akan,(**April-2014)**,“*Cyclostationary Features Based Spectrum Sensing For Cognitive Radio,*” International Journal ofScientific and Engineering Research, Volume 5, Issue 4, pp. 203–206.

HOBBIES

Reading, Video Games, Movies