Friedrich-Alexander-Universität Erlangen-Nürnberg
Lehrstuhl für Regelungstechnik
Prof. Dr.-Ing. G. Roppenecker Prof. Dr.-Ing. Th. Moor

Diplomarbeit

Hierarchical Fault Diagnosis for Discrete Event Systems: Theoretical Development and Application

Als Diplomarbeit
vorgelegt von

Tobias Barthel

Betreuer:	Betreuer:	Ausgabedatum: 19.12.08
Dr.-Ing. Klaus Schmidt	Prof. Dr.-Ing. Th. Moor	Abgabedatum: 19.06.09

Tobias Barthel

Hierarchical Fault Diagnosis for Discrete Event Systems: Theoretical Development and Application

Aufgabenstellung:

The failure diagnosis for discrete event systems (DES) has been an active area of research for more than 10 years. In the general setting, it is desired to detect the occurrence of unobservable failure events by comparing partial observations of the system evolution and a model of the possibly faulty system behaviour. Several approaches to solve this diagnosis problem have been proposed in the literature.

In this context, the major objective of this thesis is the computational support of existing failure diagnosis approaches and the development of a novel abstraction-based approach that can be applied to DES that are composed of multiple subsystems.

In order to address the first task, the libFAUDES software library for DES that was developed at the Lehrstuhl für Regelungstechnik, has to be extended by a *diagnosis* plug-in for the failure diagnosis of DES. Furthermore, various examples from the literature and new application examples shall be used to verify the functionality of this diagnosis plug-in.

The efficient abstraction-based controller synthesis techniques for discrete event systems (DES) that have been developed at the Lehrstuhl für Regelungstechnik are the basis for the second task. Analogous sufficient conditions for the abstraction-based failure diagnosis have to be established. Furthermore, the applicability of these conditions to practical systems has to be verified using a Fischertechnik model of a manufacturing system that is available at the Lehrstuhl für Regelungstechnik. In addition, algorithmic support for the abstraction-based failure diagnosis of DES has to be included in the designed diagnosis plug-in.

Es wird ausdrücklich auf die „Richtlinien zur Anfertigung von Studien- und Diplomarbeiten“ hingewiesen.

(Prof. Dr.-Ing. Th. Moor) (Dr.-Ing. Klaus Schmidt)

Tobias Barthel

Erklärung

Ich versichere, dass ich die vorliegende Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Erlangen, den 19. Juni 2009

(Tobias Barthel)

’Danke Oma Helene-Elisabeth! Dein Nusskuchen ist so wunderbar,
dass ich ihn ohne Einschränkung an meine Leser weitergebe.’

Tim Mälzer

Contents

Chapter 1

Introduction

Many of today's technical systems and processes can be modelled as discrete events systems (DES). The DES framework is capable of describing the behaviour of event driven systems such as manufacturing systems, transportation and traffic systems, as well as communication systems, to mention but a few.

About two decades ago, Ramadge and Wonham established a fundamental framework for the control of DES in their seminal paper "Supervisory control of a class of discrete event systems" [9]. They use a feedback controller denoted as *supervisor* to ensure that the system works as specified. The supervisor observes the events occurring in the system and disables the execution of events according to the specification and its control strategy. Since the supervisor is only able to record observable events, unobservable failures occurring in the system will not be noted. Thus, the system might behave in an undesired or unpredictable manner that cannot be corrected by the supervisor's interventions.

To meet this problem, fault detection and isolation became an active area of research since the 1990s. Sampath *et al.* introduced the notion of *diagnosability*, provided conditions for a language to be diagnosable and presented a systematic procedure for detection of failure events using *diagnosers* which observe the on-line behaviour of the system under investigation [11][12]. Based on the work of Sampath *et al.*, further research has been made regarding the procedure of testing a systems diagnosability and Jiang *et al.* and Yoo and Lafortune presented polynomial-time algorithms to verify diagnosability of a DES in [5] and [16], respectively.

Since complex large-scale systems are cumbersome for fault diagnostics and due to the fact that a lot of technical system exhibit a modular or decentralized structure, failure diagnosis of DESs composed of multiple subsystems has become a crucial area of interest. Different approaches were elaborated in this area. Among them, Qiu and Kumar introduced the notion of *codiagnosability* for plants that are observed by several diagnosers [8] and Zhou *et al.* introduced the notion of *modular diagnosability* that uses local diagnosers dependant on local subsystems [17]. Further advance to hierarchical fault diagnosis has been made by Su and Wonham [14] who proposed a hierarchical computational procedure and a *multiresultional diagnosis approach*. Recently, Paoli and Lafortune [6] introduced L_1-*diagnosability* with the objective of detecting failure events using only high-level observations.

In this thesis, a novel approach to decentralized diagnosability is developed. We present an abstraction-based method and supply a computational implementation to test the diagnosability of a system consisting of several subsystems with local specifications, where neither the construction of the overall system nor the overall specification is needed.

Based on the basics about formal languages and automata that are presented in Chapter 2, diagnostics with respect to failure events is introduced in Chapter 3. The notions of *dignosability* and *I-diagnosability* are explained and the construction of a basic diagnoser that can be used for online diagnosis is presented. A method and algorithm for testing diagnosability is adopted from [5] and modified for the application to I-diagnosability as well. Finally we present the algorithmic implementation of the diagnoser structure and the integration of the diagnosability tests into a *diagnosis* plug-in of the libFAUDES software library. (For a short introduction to libFAUDES, see Chapter 3.4.)

In Chapter 4, we state the notion of *language diagnosability* following [8] which defines the diagnosability of a system with respect to a specification language and develop a method to test it. This method is also implemented using in the diagnosis plug-in.

Based on this, we investigate decentralized diagnosis in Chapter 5. We introduce the notion of the *loop-preserving observer* and therewith establish a sufficient condition for decentralized diagnosability of individual subsystems which is then generalized to decentralized diagnosability of a overall system. Furthermore, we show that our conditions are a useful approach to test decentralized diagnosability by illustrating that the violation of these conditions also leads to the violation of diagnosability in practical cases. The chapter is concluded with the implementation of the decentralized diagnosis test in the diagnosis plug-in of libFAUDES.

The applicability of the newly developed method is demonstrated in Chapter 6. We use two subsystems of a Fischertechnik model of a manufacturing system and perform the test of decentralized diagnosis using the extended software library.

A conclusion completes the thesis and gives perspectives for possible fields of further research.

Chapter 2

Basic Notions and Definitions of Discrete Event Systems

This chapter provides the basic notions and definitions of discrete events systems (DES) that are essential to understand the following chapters. The presented concepts are mainly adopted from [3], and the reader is referred to this reference for further information.

A DES has got a *discrete* state space and an *event-driven* state transition mechanism. The discrete states only change at discrete points in time, and only due to the asynchronous occurrence of discrete events which are not triggered by time.

A formal way of describing the behaviour of a DES are *languages*.

2.1 Languages

We denote the finite event set $\Sigma = \{\sigma_1, \sigma_2, \ldots, \sigma_m\}$ of a DES as an *alphabet*. A sequence of events taken from this alphabet is called *string* or *trace*, and the *empty string* is denoted by ε. If s is a string, the number of events contained in s (counting multiple occurrences of the same event) is called the *length of s* and is denoted by $|s|$.

Definition 2.1 (Language). A language defined over an event set Σ is a set of finite-length strings formed from events in Σ.

Definition 2.2 (Kleene-Closure). The set of all finite strings of elements of Σ, including the empty string ε is called *Kleene-closure* of Σ and denoted by Σ^*.

In the scope of this thesis, several operations on languages as defined in Definition 2.1 are needed. In addition to the usual set operation, such as union, intersection, and difference with respect to Σ^* the following operations are relevant:

- *Concatenation:* Let $L_a, L_b \subseteq \Sigma^*$, then $L_aL_b := \{s \in \Sigma^* \mid s = s_as_b,\ s_a \in L_a,\ s_b \in L_b\}$.
- *Complement:* Let $L \subseteq \Sigma^*$, then the complement of L is defined as $L^c = \Sigma^* - L$.
- *Prefix-Closure:* Let $L \subseteq \Sigma^*$, then the prefix-closure $\overline{L}$ of L is defined as $\overline{L} := \{s \in \Sigma^* \mid \exists t \in \Sigma^*$ such that $st \in L\}$. L is said to be *prefix-closed* if $L = \overline{L}$.
- *Kleene-closure:* Let $L \subseteq \Sigma^*$, then $L^* := \{\varepsilon\} \cup L \cup LL \cup LLL \ldots$
- *Post-language:* Let $L \subseteq \Sigma^*$ and $s \in L$. Then the post-language of L after s, denoted by L/s, is the language $L/s := \{t \in \Sigma^* \mid st \in L\}$. By definition, $L/s = \emptyset$ if $s \notin \overline{L}$.

Next, the *natural projection* is defined according to [10]. This function erases all events from a string $s \in \Sigma^*$ that do not belong to a given alphabet $\hat{\Sigma}$.

Definition 2.3 (Natural Projection). Given an observation alphabet $\hat{\Sigma} \subseteq \Sigma$ the *natural projection* $p : \Sigma^* \to \hat{\Sigma}^*$ is defined by

$$\begin{aligned} p(\varepsilon) &:= \varepsilon \\ p(\sigma) &:= \begin{cases} \sigma & \text{if } \sigma \in \hat{\Sigma} \\ \varepsilon & \text{if } \sigma \notin \hat{\Sigma} \end{cases} \\ p(s\sigma) &:= p(s)p(\sigma) \quad \text{for } s \in \Sigma^* \text{ and } \sigma \in \Sigma. \end{aligned}$$

Given a string from the smaller alphabet $\hat{s} \in \hat{\Sigma}$, the *inverse projection* returns the set of all strings $s \in \Sigma$ in the larger alphabet that project, with p, to the given string $\hat{s}$.

Definition 2.4 (Inverse Projection). Given $\hat{\Sigma} \subseteq \Sigma$, the *inverse projection* $p^{-1} : \hat{\Sigma}^* \to 2^{\Sigma^*}$ is defined as

$$p^{-1}(\hat{s}) := \{s \in \Sigma^* \mid p(s) = \hat{s}\}.$$

The projections p and their inverses p^{-1} can be extended to languages by applying them to all strings in the language. For $L \subseteq \Sigma^*$ the natural projection is defined as

$$p(L) := \{t \in \hat{\Sigma} \mid \exists\, s \in L \text{ such that } p(s) = t\}$$

and for $L \subseteq \hat{\Sigma}$ the inverse projection is

$$p^{-1}(L) := \{s \in \Sigma \mid \exists t \in L \text{ such that } p(s) = t\}.$$

Note that $p\left[p^{-1}(L)\right] = L$ but in general $L \subseteq p^{-1}\left[p(L)\right]$.

As it will be needed later in Chapter 5, we define the *natural observer* as in [4].

Definition 2.5 (Natural Observer). Consider the natural projection $p : \Sigma^* \to \hat{\Sigma}^*$, where $\hat{\Sigma} \subseteq \Sigma$, and a regular language $L \subseteq \Sigma^*$. p is called an *L-observer* if for all $t \in p(L)$ and for all $s \in \overline{L}$ it holds that

$$p(s) \leq t \quad \Rightarrow \quad \exists u \in \Sigma^* \text{ such that } su \in L \text{ and } p(su) = t.$$

This definition states that if the projection string $p(s)$ can be extended to t in the smaller alphabet, then there has to exist an extension of s in the original alphabet so that the extended string in the original alphabet projects to the extension of $p(s)$ in the smaller alphabet. This observer condition ensures that the system that is observed by the projection p will not reach a state with a different future than the observed one.

Since it is not always easy and practical to enumerate all strings in a language, *automata* are introduced as a framework for constructing, representing and manipulating languages.

2.2 Automata

An automaton is a structure capable of representing languages according to well-defined rules. First, the definition of the nondeterministic automaton is presented.

Definition 2.6 (Nondeterministic Automaton). A *nondeterministic automaton* G_{nd} is a five-tuple

$$G_{\text{nd}} = (X, \Sigma \cup \{\varepsilon\}, \delta_{\text{nd}}, X_0, X_{\text{m}}),$$

where X is the set of states, $\Sigma \cup \{\varepsilon\}$ denotes the finite set of events including the empty string, $\delta_{\text{nd}} : X \times \Sigma \cup \{\varepsilon\} \to 2^X$ is the transition function, $X_0 \subseteq X$ is the set of initial states, and $X_{\text{m}} \subseteq X$ is the set of marked states.

The transition function δ_{nd} is in general a partial function on its domain which means that for every $x \in X$, δ_{nd} is only defined for a subset of the alphabet Σ. In the following we write $\delta_{\text{nd}}(x,\sigma)!$ to denote that $\delta_{\text{nd}}(x,\sigma)$ is defined. The *active event function* $\Gamma : X \to 2^{\Sigma}$, which is the set of all events σ for which $\delta_{\text{nd}}(x,\sigma)$ is defined, is omitted in this definition because it can easily be derived from δ_{nd}. (Given a set A the notion 2^A means the power set of A, i. e., the set of all subsets of A.)

For convenience, δ_{nd} is extended to the domain $X \times \Sigma^*$, so that it applies to a string u as well.

$$\delta_{\text{nd}}(x, u\sigma) := \{z \mid z \in \delta_{\text{nd}}(y,\sigma) \text{ for some state } y \in \delta_{\text{nd}}(x,u)\}.$$

Considering all directed traces of a generator, starting from initial states, and among these only those that end in a marked state, we can now define the *generated* and *marked language* as the connection between automata and languages.

Definition 2.7 (Generated and Marked Language). The *generated language* of G_{nd} is

$$L(G_{\mathrm{nd}}) = \{s \in \Sigma^* \mid \exists x \in x_0 \text{ such that } \delta_{\mathrm{nd}}(x,s)!\}.$$

The *marked language* of G is

$$L_{\mathrm{m}}(G_{\mathrm{nd}}) = \{s \in L(G_{\mathrm{nd}}) \mid \exists x \in x_0 \text{ such that } \delta_{\mathrm{nd}}(x,s) \cap X_{\mathrm{m}} \neq \emptyset\}.$$

A frequently used special case of the nondeterministic automaton is the deterministic automaton. It does not have ε-transitions, and is has just one initial state. Furthermore its transition function δ maps to a unique successor state.

Definition 2.8 (Deterministic Automaton). A *deterministic automaton* G is a five-tuple

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

where the entries have got the same interpretation as in the definition of the nondeterministic automaton, with the following two differences:

1. δ is a function $\delta : X \times \Sigma \to X$, which means that in a state $x \in X$ an event $\sigma \in \Sigma$ will only cause a transition to a unique state in X.
2. The *initial state* $x_0 \in X$ is just a single state, and no longer a set of states.

δ is also recursively extended from domain $X \times \Sigma$ to $X \times \Sigma^*$, so that it applies to strings as well:

$$\begin{aligned} \delta(x,\varepsilon) &:= x \\ \delta(x,s\sigma) &:= f(f(x,s),\sigma) \quad \text{for } s \in \Sigma^* \text{ and } \sigma \in \Sigma. \end{aligned}$$

The languages generated and marked by the deterministic automaton G are defined as:

$$\begin{aligned} &L(G) := \{s \in \Sigma^* \mid \delta(x_0,s)!\} \\ &L_{\mathrm{m}}(G) := \{s \in L(G) \mid \delta(x_0,s) \in X_{\mathrm{m}}\}. \end{aligned}$$

A language is said to be *regular* if it can be marked by a finite-state automaton.

From the definitions of G, $L(G)$ and $\overline{L_{\mathrm{m}}(G)}$ we have that

$$L_{\mathrm{m}}(G) \subseteq \overline{L_{\mathrm{m}}(G)} \subseteq L(G).$$

Definition 2.9 (Blocking). An automaton G is said to be *blocking* if

$$\overline{L_{\mathrm{m}}(G)} \subset L(G)$$

where the set inclusion is proper, and *nonblocking* when

$$\overline{L_{\mathrm{m}}(G)} = L(G).$$

If an automaton is blocking, a *deadlock* or *livelock* can happen. A reachable state x of an automaton G is called a *deadlock* if $\Gamma(x) = \emptyset$ but $x \notin X_\mathrm{m}$. A *livelock* is a set of unmarked states of G that forms a strongly connected component (i. e., the states are reachable from one another), but with no transitions going out of the set.

We now introduce several operations on automata that are essential for the concepts discussed in this thesis.

The *parallel composition* represents the joint behaviour of two automata $G_1 = (X_1, \Sigma_1 \cup \{\varepsilon\}, \delta_1, X_{01}, X_{\mathrm{m}1})$ and $G_2 = (X_2, \Sigma_2 \cup \{\varepsilon\}, \delta_2, X_{02}, X_{\mathrm{m}2})$ that are synchronized by means of their shared events. Thus, a *shared event* $\sigma \in \Sigma_1 \cap \Sigma_2 \cup \{\varepsilon\}$ can only be executed, if the two automata both execute it at the same time. All other events can occur whenever possible.

Definition 2.10 (Parallel Composition of Nondeterministic Automata). The *parallel composition* of G_1 and G_2 is the automaton

$$G_1 \parallel G_2 := (X_1 \times X_2, \Sigma_1 \cup \Sigma_2 \cup \{\varepsilon\}, \delta_\parallel, X_{01} \times X_{02}, X_{\mathrm{m}1} \times X_{\mathrm{m}2})$$

where

$$\delta_\parallel((x_1,x_2),\sigma) := \begin{cases} \delta_1(x_1,\sigma) \times \delta_2(x_2,\sigma) & \text{if } \sigma \in \Gamma_1(x_1) \cap \Gamma_2(x_2) \\ \delta_1(x_1,\sigma) \times \{x_2\} & \text{if } \sigma \in \Gamma_1(x_1) \setminus \Sigma_2 \\ \{x_1\} \times \delta_2(x_2,\sigma) & \text{if } \sigma \in \Gamma(x_2) \setminus \Sigma_1 \\ \text{undefined} & \text{otherwise.} \end{cases}$$

In the case of deterministic automata G_1 and G_2, the parallel composition simplifies to the following definition.

Definition 2.11 (Parallel Composition of Deterministic Automata). The *parallel composition* of G_1 and G_2 is the automaton

$$G_1 \parallel G_2 := (X_1 \times X_2, E_1 \cup E_2, \delta, (x_{01}, x_{02}), X_{\mathrm{m}1} \times X_{\mathrm{m}2})$$

where

$$\delta((x_1,x_2),\sigma) := \begin{cases} (\delta_1(x_1,\sigma), \delta_2(x_2,\sigma)) & \text{if } \sigma \in \Gamma_1(x_1) \cap \Gamma_2(x_2) \\ (\delta_1(x_1,\sigma), x_2) & \text{if } \sigma \in \Gamma_1(x_1) \setminus \Sigma_2 \\ (x_1, \delta_2(x_2,\sigma)) & \text{if } \sigma \in \Gamma(x_2) \setminus \Sigma_1 \\ \text{undefined} & \text{otherwise.} \end{cases}$$

The parallel composition of languages can be derived from the parallel composition of automata as follows:

$$\begin{aligned} L(G_1) \parallel L(G_2) &= L(G_1 \parallel G_2) \\ L_\mathrm{m}(G_1) \parallel L_\mathrm{m}(G_2) &= L_\mathrm{m}(G_1 \parallel G_2). \end{aligned}$$

Chapter 3

Diagnosis with respect to Failure Events

In this chapter, diagnosability of DES with respect to failure events is presented. The basic concern is to identify the occurrence, if any, of certain unobservable failure events in a DES. Therefore, all occurring observable events are tracked and the actual state of the system is estimated.

3.1 The Notion of Diagnosability

In [12], Sampath *et al.* introduce two related notions of diagnosability of DES: diagnosability and I-diagnosability. They present a systematic procedure for detection and isolation of failure events using *diagnosers* and provide necessary and sufficient conditions for a language to be diagnosable. A diagnoser is a finite state automation built from the finite state model of the observed system. It performs diagnosis while observing the on-line behaviour of the system. The diagnoser states carry failure information and thus, inspecting these states, occurrences of failures can be detected on-line with a finite delay if the system is diagnosable. In contrast, the verification of the diagnosability property of a system has to be performed off-line.

Before being able to define diagnosability according to [12], the following notions have to be introduced. Let $G = (X, \Sigma, \delta, x_0)$ be a deterministic finite state automaton. (In following, we often consider finite state automata, where all states are marked. In that case, we do not include the marked state in the description explicitly.) The event set Σ of G is partitioned as

$$\Sigma = \Sigma_{\mathrm{o}} \cup \Sigma_{\mathrm{uo}}$$

where Σ_{o} represents the set of *observable* events and Σ_{uo} represents the set of *unobservable* events. The *observable* events may be commands issued by the controller, sensor readings directly after the execution of the above commands, and changes of sensor readings. The *unobservable* events may be failure events or other events that cause changes in the system not recorded by sensors. $\Sigma_f \subseteq \Sigma$ denotes the set of failure events which are to be diagnosed. Without loss of generality, it is assumed that $\Sigma_f \subseteq \Sigma_{\text{uo}}$ since observable events can be trivially diagnosed. It may be impossible to diagnose uniquely every possible fault or one may simply be interested in knowing if one of a set of failure events occurred (e. g., if the effect of a set of failures on the system is the same). Hence, the set of failure events Σ_f is partitioned into disjoint sets corresponding to different failure types

$$\Sigma_f = \Sigma_{f1} \cup \cdots \cup \Sigma_{fm}. \tag{3.1}$$

$\Pi_f = \{1, \ldots, m\}$ denotes the index set enumerating the partitions. In the following, the expression "a failure of type F_i has occurred" will mean that some event from the set Σ_{fi} has occurred.

Furthermore, the following assumptions are met for the system under investigation.

Assumption 3.1 (Liveness). $L(G)$ is live, i. e., there is a transition defined at each state x in X. This assumption is made for the sake of simplicity.

Assumption 3.2 (No Unobservable Cycles). There does not exist any cycle of unobservable events in G, i. e., $\exists n_{\text{o}} \in \mathbb{N}$ such that $\forall ust \in L,\ s \in \Sigma_{\text{uo}}^* \ \Rightarrow\ |s| \leq n_{\text{o}}$. This ensures that G does not generate arbitrarily long sequences of unobservable events which would violate diagnosability.

Let p_{o} be the projection $p_{\text{o}} \colon \Sigma^* \to \Sigma_{\text{o}}^*$, and with $y \in \Sigma^*$, $p_L^{-1}(y) = \{s \in L \mid p_{\text{o}}(s) = y\}$ be the inverse projection on the language L. s_f is the final event of trace s and the set of all traces of L that end in a failure event belonging to the class Σ_{fi} is defined as

$$\Psi(\Sigma_{fi}) = \{s\sigma_f \in L \mid \sigma_f \in \Sigma_{fi}\}.$$

Given $\sigma \in \Sigma$ and $s \in \Sigma^*$, the notation $s \in \sigma$ denotes that σ is an event in s. If there exists a $\sigma_f \in \Sigma_{fi}$ such that $\sigma_f \in s$, the notion $\Sigma_{fi} \in s$ states with slight abuse of notation that $\overline{s} \cap \Psi(\Sigma_{fi}) \neq \emptyset$, where $\overline{s}$ is the prefix-closure of s. Additionally, X_{o} is defined as

$$X_{\text{o}} = \{x_0\} \cup \{x \in X \mid \text{there is an observable transition leading to } x\}. \tag{3.2}$$

3.1.1 Diagnosability

With the definitions and notions introduced above it is now possible to define diagnosability formally. Roughly speaking, a language L is diagnosable if, using the record of observed events, it is possible to detect the occurrence of failures of any type with a finite delay.

Definition 3.1 (Diagnosability). A prefix-closed and live language L is said to be *diagnosable* with respect to the projection p_{o} and with respect to the partition Π_f if the following holds

$$(\forall i \in \Pi_f)(\exists n_i \in \mathbb{N})[\forall s \in \Psi(\Sigma_{fi})](\forall t \in L/s) \quad [|t| \geq n_i \Rightarrow D]$$

where the diagnosability condition D is

$$\omega \in p_L^{-1}\left[p_{\text{o}}(st)\right] \ \Rightarrow\ \Sigma_{fi} \in \omega.$$

This definition means: Given a trace $s \in L$ that ends in a failure event from the set Σ_{fi}, and any sufficiently long continuation t of s. The diagnosability condition D requires that every trace $\omega \in L$ that produces the same record of observable events as st should contain a failure event from the set Σ_{fi}. Thus, along every continuation t of s the occurrence of a failure of type F_i can be detected in at most n_i transitions after s.

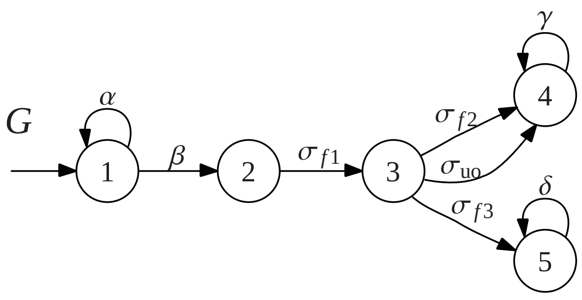


Figure 3.1: Example illustrating diagnosability.

Figure 3.1 depicts a system to illustrate diagnosability. Here, α,β,γ, and δ are observable events, while σ_{uo} is an unobservable event and σ_{f1},σ_{f2}, and σ_{f3} represent unobservable failure events. If the failure partition is chosen with $\Sigma_{f1} = \{\sigma_{f1},\sigma_{f2}\}$ and $\Sigma_{f2} = \{\sigma_{f3}\}$, i. e., it is not required to distinguish between failures σ_{f1} and σ_{f2}, then the system is diagnosable with $n_1 = 2$ and $n_2 = 1$. On the other hand, if the failure partition is $\Sigma_{f1} = \{\sigma_{f1}\}$, $\Sigma_{f2} = \{\sigma_{f2}\}$, and $\Sigma_{f3} = \{\sigma_{f3}\}$, then the system is not diagnosable because it is not possible to identify the occurrence of failure σ_{f2}.

3.1.2 I-Diagnosability

I-diagnosability, as presented by Sampath *et al.* in [12], is a relaxed definition of diagnosability that requires the diagnosability condition D only to hold for traces in which the failure event is followed by certain observable *indicator* events associated with the corresponding failure type. This is useful for systems where a component or a part of the system, respectively, might fail while the rest of the system is still able to execute an arbitrarily long non-faulty trace without being able to recognize the failure. If, e. g., in an HVAC (heating, ventilating, and air conditioning) system a valve fails, one might not be able to tell so until the controller tries to operate the valve. To meet this problem, I-diagnosability requires detection of failures only after the occurrence of an indicator event corresponding to the failure. Spoken in the example of the valve, the indicator events could be the commands *open valve* and *close valve*.

Let $\Sigma_I \subseteq \Sigma_o$ denote the set of indicator events, and let $I_f : \Sigma_f \to 2^{\Sigma_I}$ denote the indicator map. The failure event set is partitioned as in (3.1), with the additional constraint that for each $i = 1,\ldots,m$

$$\sigma_{f1},\sigma_{f2} \in \Sigma_{fi} \;\Rightarrow\; I_f(\sigma_{f1}) = I_f(\sigma_{f2}).$$

The indicator map I is extended to failure event sets by defining

$$I(\Sigma_{fi}) = I_f(\sigma_f) \quad \text{for any} \quad \sigma_f \in \Sigma_{fi}.$$

Hence, a set of observable indicator events $I(\Sigma_{fi})$ is associated with each failure type F_i.

Definition 3.2 (I-Diagnosability). A prefix-closed and live language L is said to be *I-diagnosable* with respect to the projection p_o, the failure partition Π_f on Σ_f, and the indicator map I if the following holds

$$(\forall i \in \Pi_f)(\exists n_i \in \mathbb{N})[\forall s \in \Psi(\Sigma_{fi})](\forall t_1 t_2 \in L/s \mid st_1 \in \Psi[I(\Sigma_{fi})])$$
$$[|t_2| \geq n_i \Rightarrow D]$$

where the diagnosability condition D is

$$\omega \in p_L^{-1}[p_o(st_1t_2)] \;\Rightarrow\; \Sigma_{fi} \in \omega.$$

Here, $\Psi[I(\Sigma_{fi})]$ denotes the set of all traces of L that end in an indicator event from the set Σ_{fi}. It is required, that the occurrence of a failure event of the type F_i which is "some when" followed by an indicator event from the set $I(\Sigma_{fi})$ should be detected in at most n_i transitions of the system after the occurrence of the indicator event. So if the failure event occurs, but is not followed by a matching indicator event, I-diagnosability does not require it to be detected.

For an illustration of I-diagnosability, consider the system represented in Figure 3.1. Given a failure partition $\Sigma_{f1} = \{\sigma_{f1}\}$, $\Sigma_{f2} = \{\sigma_{f2}\}$, and $\Sigma_{f3} = \{\sigma_{f3}\}$ and the indicator events $I(\Sigma_{f1}) = \{\gamma\}$ and $I(\Sigma_{f2}) = I(\Sigma_{f3}) = \{\delta\}$. Then the system is I-diagnosable with $n_1 = 0$ and $n_3 = 0$. Note that in this case I-diagnosability does not require the failure σ_{f2} to be identified because the corresponding indicator event δ does not follow the failure event.

3.2 The Diagnoser

We now describe the concept of a *diagnoser*. This automaton is used to perform diagnostics while observing the on-line behaviour of a system G which includes all relevant failure events in its modelling. Sampath *et al.* show in [12, Section V] that this diagnoser is adequate for on-line diagnosis of diagnosable and I-diagnosable systems, with or without multiple failures. Since the concept of multiple failures is only relevant for their diagnosability test (which we are not going to use), we omit it here.

In the following, the construction procedure of the diagnoser according to [12] is presented.

Given the label N meaning *normal*, i. e., no failure occurred, the label A meaning *ambiguous* (as explained on p. 13), and F_i with $i \in \{1, \ldots, m\}$ meaning a failure of type F_i has occurred. Then the set of failure labels is defined as $\Delta_f = \{F_1, F_2, \dots, F_m\}$ where $\left|\Pi_f\right| = m$ and the complete set of possible combinations of labels is $\Delta = \{N\} \cup 2^{\Delta_f \cup \{A\}}$.

Furthermore, with X_o from (3.2), we define $Q_\text{o} = 2^{X_\text{o} \times \Delta}$.

Now the diagnoser for the system G is the finite state automaton

$$G_\text{diag} = (Q_\text{diag}, \Sigma_\text{o}, \delta_\text{diag}, q_0^\text{diag})$$

where $Q_\text{diag}, \Sigma_\text{o}, \delta_\text{diag}$, and q_0^diag are interpreted as usual. The initial state q_0^diag is defined to be $(x_0, \{N\})$ and the transition function δ_diag is constructed as explained below. The state space Q_diag is the resulting subset of Q_o composed of the states of the diagnoser that are reachable from Q_o under δ_diag. Since the state space Q_diag of the diagnoser is a subset of Q_o, a state q_d of G_diag is of the form

$$q_d = \{(x_1, l_1), \dots, (x_n, l_n)\}$$

where $x \in X_\text{o}$ and $l_i \in \Delta$, i. e., l_i is of the form $l_i = \{N\}$, $l_i = \{A\}$, $l_i = \{F_{i_1}, F_{i_2}, \dots, F_{i_k}\}$, or $l_i = \{A, F_{i_1}, F_{i_2}, \dots, F_{i_k}\}$ where in the last two cases $\{i_1, i_2, \dots, i_k\} \subseteq \{1, 2, \dots, m\}$. The states of the diagnoser G_diag carry labelled state estimates of the observed system. The labels carry failure information and failures are diagnosed by checking these labels. The initial state is defined as $q_0^\text{diag} = \{(x_0, \{N\})\}$.

Before being able to define the transition function δ_diag, we introduce the following functions.

Let $L_\text{o}(G, x)$ denote the set of all traces that start at a state x and end at the first observable event:

$$L_\text{o}(G, x) = \{s \in L(G, x) \mid s = u\sigma,\ u \in \Sigma_\text{uo}^*,\ \sigma \in \Sigma_\text{o}\},$$

where $L(G, x)$ is the set of all traces that originate from state x in G.

Definition 3.3 (Label Propagation Function). Given $x \in X_\text{o}$, $l \in \Delta$, and $s \in L_\text{o}(G, x)$. The *label propagation function* $LP : X_\text{o} \times \Delta \times \Sigma^* \to \Delta$ propagates the label l over s, starting from x and following the dynamics of G, i. e., according to $L(G, x)$. It is defined as follows

$$LP(x, l, s) = \begin{cases} \{N\} & \text{if } l = \{N\} \wedge \forall i[\Sigma_{fi} \notin s] \\ \{A\} & \text{if } l = \{A\} \wedge \forall i[\Sigma_{fi} \notin s] \\ \{F_i \mid F_i \in l \vee \Sigma_{fi} \text{ in } s\} & \text{otherwise.} \end{cases}$$

Definition 3.4 (Range Function). The *range function* $R : Q_\text{o} \times \Sigma_\text{o} \to Q_\text{o}$ is defined as

$$R(q, \sigma) = \{\delta(x, s), LP(x, l, s) \mid (x, l) \in q \wedge s \in L_\sigma(G, x)\}$$

Definition 3.5 (Label Correction Function). The *label correction function* $LC : Q_\text{o} \to Q_\text{o}$ is defined as

$$\begin{aligned} LC(q) = \{&(x, l) \in q \mid x \text{ appears only once in all the pairs in } q\} \cup \\ &\{(x, \{A\} \cup l_{i1} \cap \dots \cap l_{ik}) \text{ whenever there exist} \\ &\text{two or more pairs } (x, l_{i1}), \dots, (x, l_{ik}) \text{ in } q\}. \end{aligned}$$

The label correction function assigns the diagnoser state labels. The label acquired by any state x along a trace s indicates the occurrence or non-occurrence when the system moves along trace s and transitions into state x.

The label A has to be interpreted as follows. Suppose that for some state $q \in Q_{\text{diag}}$ there exist two pairs $(x,l),(x,l')$ in $R(q,\sigma)$. This implies that the state x could have resulted from a failure event of a particular type, say F_i, or not. In this case we attach the label A to denote that there is an ambiguity. Hence, label A has to be interpreted as meaning "either F_i or not F_i" for $i \in \{1,\ldots,m\}$.

Now the transition function $\delta_{\text{diag}}\colon Q_0 \times \Sigma_0 \to Q_0$ is defined as

$$q_2 = \delta_{\text{diag}}(q_1,\sigma) = LC\,[R(q_1,\sigma)]$$

with $\sigma \in e_d(q_1)$ where $e_d(q_1)$ is the active event set of G_{diag} at the state q_1:

$$e_d(q_1) = \bigcup_{(x,l)\in q_1} \{P(s) \mid s \in L_{\text{o}}(G,x)\}.$$

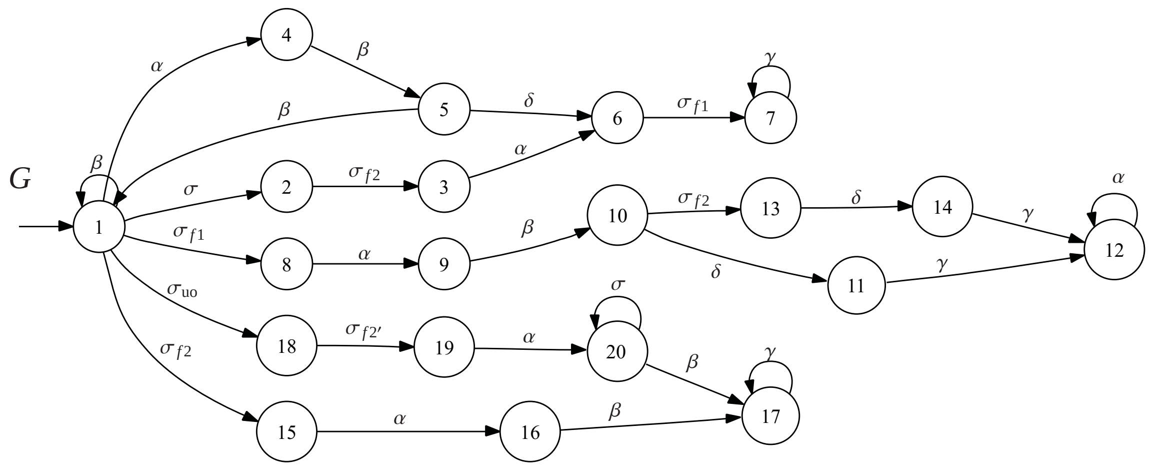
To illustrate the construction of the diagnoser Figure 3.2 depicts an example of a system G and its diagnoser G_{diag}. Here $\alpha,\beta,\gamma,\delta$, and σ are observable events while $\sigma_{\text{uo}},\sigma_{f1},\sigma_{f2}$, and $\sigma_{f2'}$ are unobservable. The failure partition is $\Sigma_{f1} = \{\sigma_{f1}\}$ and $\Sigma_{f2} = \{\sigma_{f2},\sigma_{f2'}\}$. In the following a state-label-pair (x,l) will be represented as xl for the sake of clarity.

3.3 Diagnosability Testing

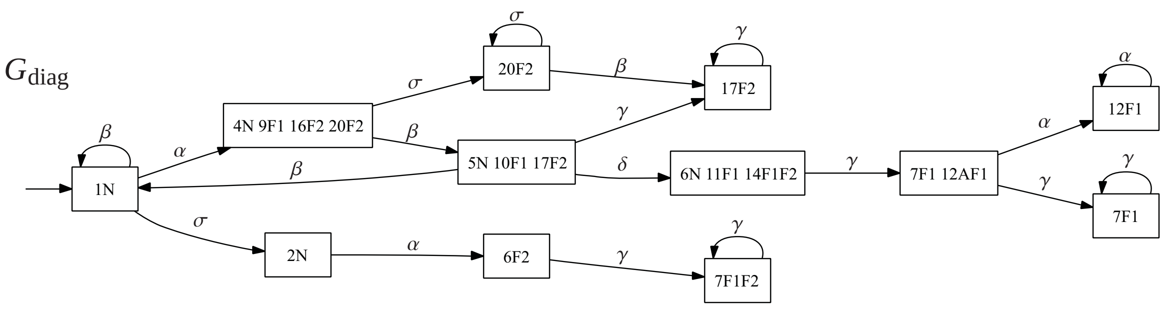
In [12], Sampath *et al.* proposed a necessary and sufficient condition to test diagnosability. Their testing procedure requires the diagnoser to be constructed first and then checks for ambiguous states and *F_i-indeterminate* cycles in the diagnoser. In simple terms, the latter are cycles of *F_i-uncertain* states (i. e., states of the diagnoser that contain state estimates of the same state, the one containing F_i in its label, the other not) for which there exists (i) a corresponding cycle of states in the original generator that carry F_i in their labels in the cycle of the diagnoser and (ii) a corresponding cycle of states in the original generator that do not carry F_i in their labels in the cycle of the diagnoser.

The major disadvantage of the condition presented by Sampath *et al.* that in order to test diagnosability of a system the diagnoser has to be constructed first and the state space of the diagnoser is in the worst case exponential in the cardinality of the state space of the system model. Practically spoken, constructing the diagnoser is cumbersome if afterwards it turns out that the system is not diagnosable.

To meet this problem, Jiang *et al.* [5] and Yoo and Lafortune [16] proposed two different tests of diagnosability that require only polynomial time in the number of states of the system model.



(a) The System G.



(b) The Diagnoser G_{diag}.

Figure 3.2: Example construction of the diagnoser G_{diag} from the system G [12]. The state names of G_{diag} carry state estimates for the plant G. Label N indicates "normal" and label F_i the occurrence of a failure of type F_i. If all state estimates in a diagnoser state name carry unique failure labels, then the a failure of type F_i is identified.

In the following section the diagnosability test according to Jiang *et al.* will be presented and in Section 3.3.2 we propose a modification of the approach of Jiang *et al.* with the objective of testing I-diagnosability in polynomial-time as well.

3.3.1 Testing Diagnosability

In [5], Jiang *et al.* presented an approach to test a system for diagnosability that does not require the construction of the diagnoser. The complexity of their method is polynomial in the number of states of the system and also polynomial in the number of failure types.

In the following we will demonstrate the algorithm proposed by Jiang *et al.* The system G under investigation has to meet the assumptions stated in Chapter 3.1 and additionally to the known notions we define a *path* in G as a sequence of transitions $(x_1, \sigma_1, x_2, \ldots, \sigma_{n-1}, x_n)$ such that $\delta(x_i, \sigma_i)$ exists with $\delta(x_i, \sigma_i) = x_{i+1}$ for all $i \in \{1, \ldots, n-1\}$. A path is called a *cycle* if $x_n = x_1$. Let $\mathcal{F} = \{F_i \mid$

$i = 1,2,\dots,m\}$ denote the set of failure types, $\psi : \Sigma \to \mathcal{F}$ be the failure assignment function for each event $\sigma \in \Sigma$, and $p_o : \Sigma^* \to \Sigma_o^*$ be the observation mask. As before, we do not include the marked state in the description of automata explicitly because we only consider finite state automata, where all states are marked.

Algorithm 3.1 (Diagnosability Test). For a given system $G = (X, \Sigma, \delta, x_0)$ with an observation mask p_o and a failure assignment function ψ, do the following:

1. Construct a nondeterministic finite state automaton $G_o = (X_o, \Sigma_o, \delta_o, x_0^o)$ with language $L(G_o) = p(L(G))$ as follows:
    - $X_o = \{(x, f) \mid x \in X_1 \cup \{x_o\}, f \subseteq \mathcal{F}\}$ is the finite set of states, where $X_1 = \{x \in X \mid \delta(x', \sigma) = x,\ \text{with } x' \in X, p_o(\sigma) \neq \varepsilon\}$ is the set of states in G that can be reached through an observable transition, and f is the set of failure types along certain paths from x_0 to x.
    - Σ_o, the set of observable events, is the set of events labels for G_o.
    - δ_o is the set of transitions. $\delta_o((x, f), \sigma) = (x', f')$ if and only if there exists a path $(x, \sigma_1, x_1, \dots, \sigma_n, x_n, \sigma, x'), n \geq 0$ in G such that $\forall i \in \{1,2,\dots,n\}, p_o(\sigma_i) = \varepsilon, p_o(\sigma) = \sigma$, and $f' = \{\psi(\sigma_i) \mid \psi(\sigma_i) \neq \emptyset, 1 \leq i \leq n\} \cup f$; otherwise $\delta_o((x, f), \sigma)$ is not defined.
    - $x_0^o = (x_0, \emptyset) \in X_o$ is the initial state.
2. Compute $G_d = G_o \parallel G_o = (X_d, \Sigma_o, \delta_d, x_0^d)$, where
    - $X_d = \{(x_1^o, x_2^o) \mid x_1^o, x_2^o \in X_o\}$ is the set of states.
    - Σ_o is the set of events labels for G_d.
    - δ_d is the transition function. $\delta_d((x_1^o, x_2^o), \sigma) := (y_1^o, y_2^o)$ if and only if $\delta_o(x_1^o, \sigma) = y_1^o$ and $\delta_o(x_2^o, \sigma) = y_2^o$; otherwise $\delta_d((x_1^o, x_2^o), \sigma)$ is not defined.
    - $x_0^d = (x_0^o, x_0^o) \in X_d$ is the initial state.
3. Check whether there exists in G_d a cycle $cl = (x_1, \sigma_1, x_2, \dots, x_n, \sigma_n, x_1),\ n \geq 1, x_i = ((x_i^1, f_i^1), (x_i^2, f_i^2)),\ i = 1,2,\dots,n,$ such that $f_1^1 \neq f_1^2$. If there exists such a cycle, then the system G is not diagnosable; otherwise it is diagnosable.

Theorem 3.1 (Diagnosability). G is diagnosable if and only if for every cycle cl in G_d,

$$cl = (x_1, \sigma_1, x_2, \dots, x_n, \sigma_n, x_1), \quad n \geq 1, \quad x_i = ((x_i^1, f^1), (x_i^2, f^2)), \quad i = 1,2,\dots,n$$

we have $f^1 = f^2$.

As it is given in [5], the proof of this theorem is omitted here.

The complexity of the method shown in Algorithm 3.1 is

$$\mathcal{O}(|X|^4 \times 2^{4|\mathcal{F}|} \times |\Sigma_o|)$$

which is polynomial in the number of states in G and exponential in the number of failure types in G. In order to make the complexity polynomial in the number of failure types as well note that a system is diagnosable with respect to the failure types $\mathcal{F} = \{F_i \mid i = 1, 2, \ldots, m\}$ if and only if it is diagnosable with respect to each individual failure type $F_i, i = 1, 2, \ldots, m$. Thus, Algorithm 3.1 can be applied m different times to test diagnosability of the system G with respect to the individual failure type set $\{F_1\}, \ldots, \{F_m\}$. Now each failure type set is a singleton (a set with just one element), the complexity of each such test is $\mathcal{O}(|X|^4 \times 2^{4|1|} \times |\Sigma_o|) = \mathcal{O}(|X|^4 \times |\Sigma_o|)$. The overall complexity of testing diagnosability of G is

$$\mathcal{O}(|X|^4 \times |\Sigma_o| \times |\mathcal{F}|)$$

which is polynomial in the number of states of the system and linear in the number of failure types. [5]

To illustrate the test according to Algorithm 3.1 consider the system shown in Figure 3.3. σ_{uo} is an unobservable event and σ_{f1}, σ_{f2} are unobservable failure events. Let $\mathcal{F} = \{F_1, F_2\}$, $\psi(\sigma_{uo}) = \psi(\sigma_i) = \emptyset, i = 1, 2, 3$, and $\psi(\sigma_{f1}) = F_1, \psi(\sigma_{f2}) = F_2$. Here, we will just consider the single failure type F_1 and thus Algorithm 3.1 is only applied for F_1. We first derive G_o from G (see Figure 3.4). For compatibility, we will label states with no failure label with N instead of the empty set. To obtain G_d, we then compute the parallel composition of G_o with itself. In G_d, as depicted in Figure 3.5, note the self loop at state (4N,4F$_1$). From the last step in Algorithm 3.1 it follows that the system G is not diagnosable with respect to the given failure partition.

However, if the failure types do not have to be distinguished, i. e., $\psi(\sigma_{f1}) = \psi(\sigma_{f2}) = F$, G_o and G_d result in the automata depicted in Figure 3.6. As there do not exist any cycles with different failure labels in G_d, the system G is now diagnosable with respect to the new failure partition.

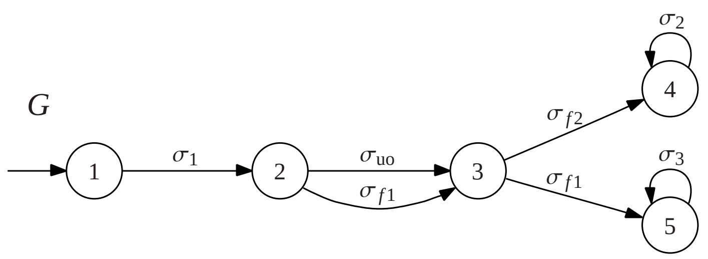


Figure 3.3: The system G.

3.3.2 Testing I-Diagnosablility

The approach of Jiang *et al.* to test a system G for diagnosability applies to diagnosability only. In order to test for I-diagnosability in a similar manner we extend their method as described below.

I-diagnosability requires the detection of a failure (within a bounded number of transition) only if one of its indicator events occurred *after* the failure event. According to [12], the condition of

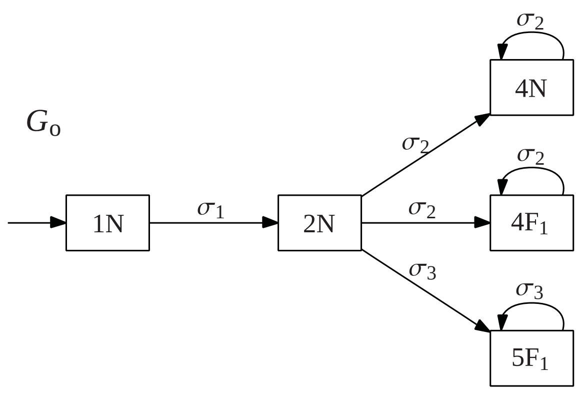


Figure 3.4: The nondeterministic automaton G_o.

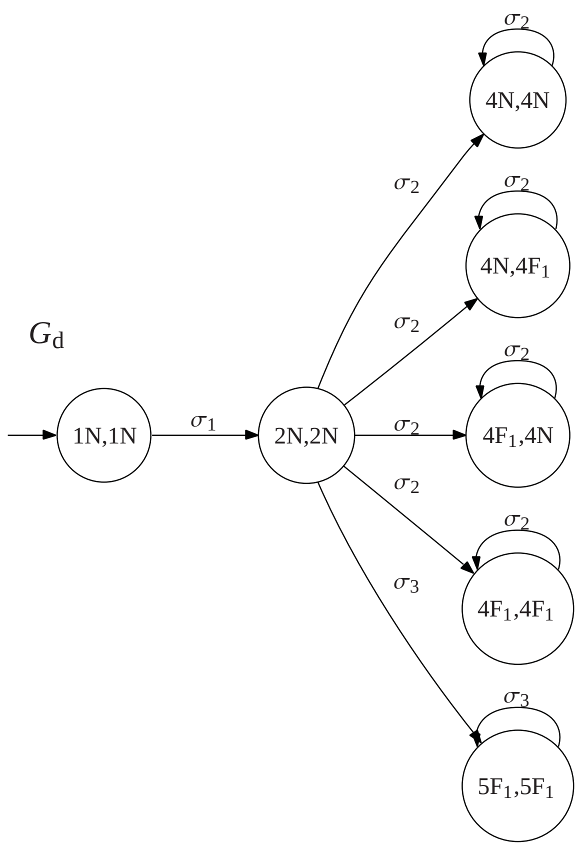
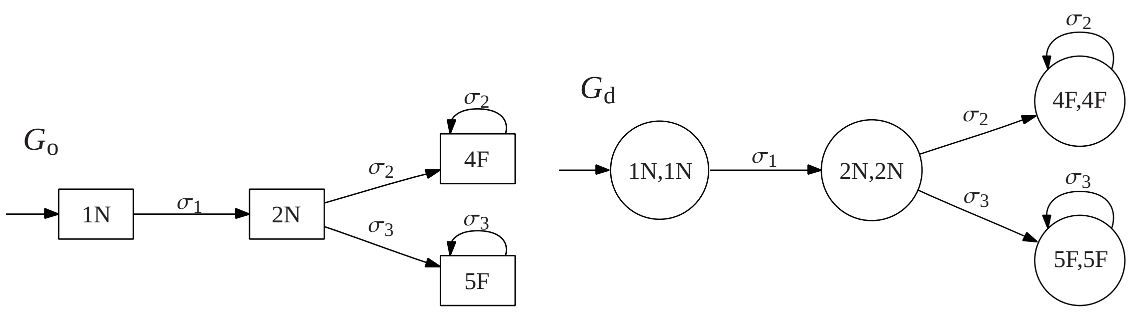


Figure 3.5: G_d.



(a) G_o. (b) G_d.

Figure 3.6: G_d for a single failure type.

I-diagnosability is violated if there exists two traces s_1 and s_2 in $L(G)$, that both have the same observable projection, and s_1 contains a failure event from the failure set Σ_{fi} followed by an indicator event for the set $I(\Sigma_{fi})$ while s_2 does not contain any event from the set Σ_{fi}.

Thus, we modify Algorithm 3.1 so that traces in G_d should only be considered after the occurrence of an indicator event that follows a failure event from the corresponding failure set. For simplicity, we only take into account an individual failure set Σ_f. To check for I-diagnosability with respect to $\mathcal{F} = \{F_i \mid i = 1, 2, \ldots, m\}$ one has to perform the following algorithm for all F_i separately.

Algorithm 3.2 (I-Diagnosability Test for Failure Type F). For a given system $G = (X, \Sigma, \delta, x_0)$ with an observation mask p_o and a failure assignment function ψ, do the following:

1. Construct a nondeterministic finite state automaton $G_o = (X_o, \Sigma_o, \delta_o, x_0^o)$ with language $L(G_o) = p(L(G))$ as follows:
   - $X_o = \{(x, f) \mid x \in X_1 \cup \{x_o\}, f \subseteq \Sigma_f\}$ is the finite set of states, where $X_1 = \{x \in X \mid \delta(x', \sigma) = x$, with $x' \in X, p_o(\sigma) \neq \varepsilon\}$ is the set of states in G that can be reached through an observable transition, and f is the set of failure types along certain paths from x_0 to x.
   - Σ_o, the set of observable events, is the set of events labels for G_o.
   - δ_o is the set of transitions. $\delta_o((x, f), \sigma) = (x', f')$ if and only if there exists a path $(x, \sigma_1, x_1, \ldots, \sigma_n, x_n, \sigma, x'), n \geq 0$ in G such that $\forall i \in \{1, 2, \ldots, n\}, p_o(\sigma_i) = \varepsilon, p_o(\sigma) = \sigma$, and $f' = \{\psi(\sigma_i) \mid \psi(\sigma_i) \neq \emptyset, 1 \leq i \leq n\} \cup f$; otherwise $\delta_o((x, f), \sigma)$ is not defined.
   - $x_0^o = (x_0, \emptyset) \in X_o$ is the initial state.
2. Compute $G_d = G_o \parallel G_o = (X_d, \Sigma_o, \delta_d, x_0^d)$, where
   - $X_d = \{(x_1^o, x_2^o) \mid x_1^o, x_2^o \in X_o\}$ is the set of states.
   - Σ_o is the set of events labels for G_d.
   - δ_d is the set of transitions. $\delta_d((x_1^o, x_2^o), \sigma) = (y_1^o, y_2^o)$ if and only if $\delta_o(x_1^o, \sigma) = y_1^o$ and $\delta_o(x_2^o, \sigma) = y_2^o$; otherwise $\delta_d((x_1^o, x_2^o), \sigma)$ is not defined.

- $x_0^{\mathrm{d}} = (x_0^{\mathrm{o}}, x_0^{\mathrm{o}}) \in X_{\mathrm{d}}$ is the initial state.

3. Check whether there exists in G_{d} a cycle

   $cl = (x_1, \sigma_1, x_2, \ldots, x_n, \sigma_n, x_1), n \geq 1, x_i = ((x_i^1, f_i^1), (x_i^2, f_i^2)), i = 1, 2, \ldots, n$, such that $\exists u_0 \in \Sigma_{\mathrm{o}}^* \Sigma_{F,I} \Sigma_{\mathrm{o}}^*$ with $\delta_{\mathrm{d}}(x_0^{\mathrm{d}}, u_0) = x_1$ and $f_{\mathrm{Ind}}^1 = F$ or $f_{\mathrm{Ind}}^2 = F$, with $\delta_{\mathrm{d}}(x_0^{\mathrm{d}}, u_{0,\mathrm{Ind}}) = x_{\mathrm{Ind}} = (x_{\mathrm{Ind}}^1, f_{\mathrm{Ind}}^1), (x_{\mathrm{Ind}}^2, f_{\mathrm{Ind}}^2))$ and $u_{0,\mathrm{Ind}}$ is the prefix of u_0 that ends with the indicator event,

   where $f_1^1 \neq f_1^2$. If there exists such a cycle, then the system G is not I-diagnosable for F; otherwise it is I-diagnosable for F.

Before proving that this algorithm works, we first define $I \colon \mathcal{F} \to 2^{\Sigma_I}$ as a map to the corresponding indicator events and provide the following two lemmas that Jiang *et al.* [5] derived from the definitions of G_{o} and G_{d}.

Lemma 3.1. For the state machine G_{o} it holds:

1. $L(G_{\mathrm{o}}) = p_{\mathrm{o}}(L(G))$.

2. For every path tr in G_{o} ending with a cycle,

   $$tr = ((x_0, \emptyset), \sigma_0, (x_1, f_1), \ldots, (x_k, f_k), \sigma_k, \ldots, (x_n, f_n), \sigma_n, (x_k, f_k)),$$

   we have

   - $f_i = f_j$ for any i and j in $\{k, k+1, \ldots, n\}$.
   - $\exists uv^* \in L(G)$ such that $p_{\mathrm{o}}(u) = \sigma_0 \ldots \sigma_{k-1},\ p_{\mathrm{o}}(v) = \sigma_k \ldots \sigma_n$, and $\{\psi(\sigma) \mid \sigma \in u, \psi(\sigma) \neq \emptyset\} = \{\psi(\sigma) \mid \sigma \in uv, \psi(\sigma) \neq \emptyset\} = f_k$.

Lemma 3.2. For every path tr in G_{d} ending with a cycle,

$$tr = (x_0^{\mathrm{d}}, \sigma_0, x_1, \ldots, x_k, \sigma_k, x_{k+1} \ldots, x_n, \sigma_n, x_k),$$

$x_i = ((x_i^1, f_i^1), (x_i^2, f_i^2)),\ i = 1, 2, \ldots, n$, we have

1. there exist two paths tr_1 and tr_2 in G_{o} ending with cycles, namely,

   $$tr_1 = ((x_0, \emptyset), \sigma_0, (x_1, f_1^1), \ldots, (x_k^1, f_k^1), \sigma_k, \ldots, (x_n^1, f_n^1), \sigma_n, (x_k^1, f_k^1)),$$
   $$tr_2 = ((x_0, \emptyset), \sigma_0, (x_1, f_1^2), \ldots, (x_k^2, f_k^2), \sigma_k, \ldots, (x_n^2, f_n^2), \sigma_n, (x_k^2, f_k^2)).$$

2. $f_i^1 = f_j^1$ and $f_i^2 = f_j^2$ for any i and j in $\{k, k+1, \ldots, n\}$.

Now we can define and proof I-diagnosability for a single failure type.

Proposition 3.1 (I-diagnosability of a Single Failure Type). Assume $\mathcal{F}_F$ is a singleton with $\mathcal{F}_F = \{F\}$ and let $\Sigma_F := \{\sigma \in \Sigma \mid \psi(\sigma) = F\}$ and $\Sigma_{F,I} := I(F)$.

G is I-diagnosable for F if and only if for every cycle cl in G_d with

> $cl = (x_1, \sigma_1, x_2, \ldots, x_n, \sigma_n, x_1)$, $n \geq 1$, $x_i = ((x_i^1, f^1), (x_i^2, f^2))$, $i = 1, 2, \ldots, n$ such that $\exists u_0 \in \Sigma_\mathrm{o}^* \Sigma_{F,I} \Sigma_\mathrm{o}^*$ with $\delta_\mathrm{d}(x_0^\mathrm{d}, u_0) = x_1$ and $f_\mathrm{Ind}^1 = F$ or $f_\mathrm{Ind}^2 = F$, with $\delta_\mathrm{d}(x_0^\mathrm{d}, u_{0,\mathrm{Ind}}) = x_\mathrm{Ind} = (x_\mathrm{Ind}^1, f_\mathrm{Ind}^1), (x_\mathrm{Ind}^2, f_\mathrm{Ind}^2)$ and $u_{0,\mathrm{Ind}}$ is the prefix of u_0 that ends with the indicator event,

we have $f_1 = f_2$.

Proof. For the necessity, suppose G is I-diagnosable, but there exists a cycle cl in G_d,

> $cl = (x_k, \sigma_k, x_{k+1}, \ldots, x_n, \sigma_n, x_k)$, $n \geq k$, $x_i = ((x_i^1, f^1), (x_i^2, f^2))$, $i = k, k+1, \ldots, n$ such that $\exists u_0 \in \Sigma_\mathrm{o}^* \Sigma_{F,I} \Sigma_\mathrm{o}^*$ with $\delta_\mathrm{d}(x_0^\mathrm{d}, u_0) = x_k$ and $f_\mathrm{Ind}^1 = F$ or $f_\mathrm{Ind}^2 = F$, with $\delta_\mathrm{d}(x_0^\mathrm{d}, u_{0,\mathrm{Ind}}) = x_\mathrm{Ind} = (x_\mathrm{Ind}^1, f_\mathrm{Ind}^1), (x_\mathrm{Ind}^2, f_\mathrm{Ind}^2)$ and $u_{0,\mathrm{Ind}}$ is the prefix of u_0 that ends with the indicator event,

and we have $f_1 \neq f_2$.

This implies that there exists a path tr in G_d ending with the cycle cl, i. e.,

$$tr = (x_0^\mathrm{d}, \sigma_0, x_1, \ldots, x_k, \sigma_k, x_{k+1} \ldots, x_n, \sigma_n, x_k).$$

Then from Lemma 3.2 we know that there exist two paths tr_1 and tr_2 in G_o with

$$tr_1 = ((x_0, \emptyset), \sigma_0, (x_1, f_1^1), \ldots, (x_k^1, f^1), \sigma_k, \ldots, (x_n^1, f^1), \sigma_n, (x_k^1, f^1)),$$
$$tr_2 = ((x_0, \emptyset), \sigma_0, (x_1, f_1^2), \ldots, (x_k^2, f^2), \sigma_k, \ldots, (x_n^2, f^2), \sigma_n, (x_k^2, f^2)).$$

Further, from Lemma 3.1, we have $\exists u_1 v_1^*, u_2 v_2^* \in L(G)$ such that

$$p_\mathrm{o}(u_1) = p_\mathrm{o}(u_2) = \sigma_0, \ldots, \sigma_{k+1}, \quad p_\mathrm{o}(v_1) = p_\mathrm{o}(v_2) = \sigma_k, \ldots, \sigma_n$$

and

$$\{\psi(\sigma) \mid \sigma \in u_i, \psi(\sigma) \neq \emptyset\} = \{\psi(\sigma) \mid \sigma \in u_i v_i, \psi(\sigma) \neq \emptyset\} = f_i, \quad i = 1, 2.$$

Without loss of generality, since $f_1 \neq f_2$, we assume $f_1 = \{F\}$, $f_2 = \emptyset$. Since $\exists u_0 \in \Sigma_\mathrm{o}^* \Sigma_{F,I} \Sigma_\mathrm{o}^*$ with $\delta_\mathrm{d}(x_0^\mathrm{d}, u_0) = x_k$, we can choose $u_1 = u_0$. For any integer n_k, we can choose another integer l such that $\|u_0 v_1^l\| > n_k$. Now we have $p_\mathrm{o}(u_2 v_2^l) = p_\mathrm{o}(u_0 v_1^l)$ and $\sigma_F \notin u_2 v_2^l$, $\sigma_I \in u_2 v_2^l$. This violates I-diagnosability and contradicts the hypothesis. So the necessity holds.

For the sufficiency, suppose for every cycle cl in G_d,

> $cl = (x_1, \sigma_1, x_2, \ldots, x_n, \sigma_n, x_1)$, $n \geq 1$, $x_i = ((x_i^1, f^1), (x_i^2, f^2))$, $i = 1, 2, \ldots, n$ such that $u_0 \in \Sigma_\mathrm{o}^* \Sigma_{F,I} \Sigma_\mathrm{o}^*$ with $\delta_\mathrm{d}(x_0^\mathrm{d}, u_0) = x_1$ and $f_\mathrm{Ind}^1 = F$ or $f_\mathrm{Ind}^2 = F$, with $\delta_\mathrm{d}(x_0^\mathrm{d}, u_{0,\mathrm{Ind}}) = x_\mathrm{Ind} = (x_\mathrm{Ind}^1, f_\mathrm{Ind}^1), (x_\mathrm{Ind}^2, f_\mathrm{Ind}^2)$ and $u_{0,\mathrm{Ind}}$ is the prefix of u_0 that ends with the indicator event,

it holds that $f_1 = f_2$. From Definition 3.2 we know that all such cycles satisfy I-diagnosability.

Now let $f_1 \neq f_2$. Again, without loss of generality, we assume $f_1 = \{F\}$, $f_2 = \emptyset$. From Lemma 3.2 we know that the hypothesis implies that

$$\forall x = ((x^1, f^1), (x^2, f^2)) \in X_\mathrm{d}$$

x is

(a) contained in a loop only if $\nexists u_0 \in \Sigma_\mathrm{o}^* \Sigma_{F,I} \Sigma_\mathrm{o}^*$ with $\delta_\mathrm{d}(x_0^\mathrm{d}, u_0) = x$. This case does not violate I-diagnosability.

(b) not contained in a loop and $\exists u_0 \in \Sigma_\mathrm{o}^* \Sigma_{F,I} \Sigma_\mathrm{o}^*$ with $\delta_\mathrm{d}(x_0^\mathrm{d}, u_0) = x$. The hypothesis further implies that for any state sequence $(x_1, x_2, \ldots, x_k)$ in G_d with $x_i = ((x_i^1, f_i^1), (x_i^2, f_i^2))$ for $1 \leq i \leq k$, if $f_i^1 \neq f_i^2$ for all $i \in \{1, 2, \ldots, k\}$, then the length of the state sequence is bounded by the number of states in G_d, i. e., $k \leq |X_\mathrm{d}|$.

Now let s be a trace in $L(G)$ with a F-type failure event, i. e., $\psi(s_f) = F$, we claim that $\forall v = st \in L(G)$ with $|t| > |X_\mathrm{d}| \cdot (|X| - 1)$, $\forall w \in L(G)$ with $p_\mathrm{o}(w) = p_\mathrm{o}(v)$, there is a F-type failure event contained in w.

From above, for any state $x \in X_\mathrm{d}$ that can be reached from x_0^d by executing $p_\mathrm{o}(s)$ in G_d, we have that any state sequence starting from x in G_d, a state $y = ((y^1, f^1), (y^2, f^2)) \in X_\mathrm{d}$ with $f^1 = f^2$ can be reached within $|X_\mathrm{d}| - 1$ steps. This implies that $\forall v = st \in L(G)$ with $\|p_\mathrm{o}(t)\| > |X_\mathrm{d}| - 1$, $\forall w \in L(G)$ with $p_\mathrm{o}(w) = p_\mathrm{o}(v)$, there is a F-type failure event contained in w. Further from the assumption that no unobservable cycles exist in G, each observed event in $p_\mathrm{o}(t)$ can be preceded by at most $|X| - 1$ unobserved events. It follows that for the trace t above, $|t| \leq (|p_\mathrm{o}(t)| + 1) \cdot (|X| - 1)$, i. e., $|p_\mathrm{o}(t)| \geq \frac{|t|}{|X|-1} - 1$. So if $|t| > |X_\mathrm{d}| \cdot (|X| - 1)$, then $|p_\mathrm{o}(t)| \geq \frac{|t|}{|X|-1} - 1 > \frac{|X_\mathrm{d}| \cdot (|X|-1)}{|X|-1} - 1 = |X_\mathrm{d}| - 1$, establishing our claim. It follows from 3.2 that G is I-diagnosable.

Note that we have assumed implicitly that $|X| > 1$; otherwise if $|X| = 1$, then from the assumption of no unobservable loops, no transition labelled by a failure event exists, so that the system is trivially I-diagnosable.

(c) not contained in a loop and $\nexists u_0 \in \Sigma_\mathrm{o}^* \Sigma_{F,I} \Sigma_\mathrm{o}^*$ with $\delta_\mathrm{d}(x_0^\mathrm{d}, u_0) = x$. As stated above, any state sequence $(x_1, x_2, \ldots, x_k)$ in G_d with $x_i = ((x_i^1, f_i^1), (x_i^2, f_i^2))$ for $1 \leq i \leq k$, if $f_i^1 \neq f_i^2$ $\forall i \in \{1, 2, \ldots, k\}$, is bounded by $|X_\mathrm{d}|$ before the system turns into a loop. From the assumption of liveness of G, the construction of G_d and the assumption of no unobservable loops we know that G_d is live as well. So, if no indicator event occurs before the system turns into a loop we pass on to case (a), otherwise case (b)—both in a bounded number of transitions.

So the sufficiency holds. ■

From this I-diagnosability is verified as follows.

Theorem 3.2 (I-Diagnosability). *G* is I-diagnosable if and only if *G* is I-diagnosable for all $F_i \in \mathcal{F}$.

This theorem directly follows from Proposition 3.1 by applying it to all failure types separately.

To illustrate the test of I-diagnosability as explained in Algorithm 3.2, consider the system G depicted in Figure 3.7. σ_{uo} is an unobservable event and σ_{f1}, σ_{f2} are unobservable failure events. Let $\psi(\sigma_{f1}) = F_1$, $\psi(\sigma_{f2}) = F_2$ and $I_f(\sigma_{f1}) = \sigma_{I1}$, $I_f(\sigma_{f2}) = \sigma_{I2}$.

Figure 3.8 and Figure 3.9 display the automata G_{o} and G_{d} for both failure types. In G_{d}, note that for failure type F_1 there does not exist an offending cycle as described in the last step of Algorithm 3.2 because the indicator event σ_{I1} does not occur after the failure event σ_{f1}. Thus, the system G is I-diagnosable with respect to failure type F_1.

On the other hand, for failure type F_2 there exists an offending cycle in G_{d} at state (6N,6F$_2$). So, I-diagnosability is violated for failure type F_2 and it follows from Theorem 3.2 that G is not I-diagnosable.

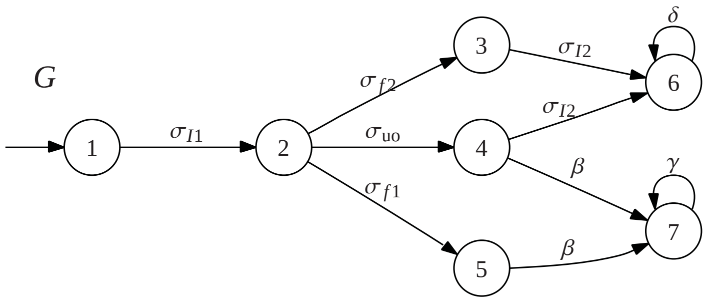


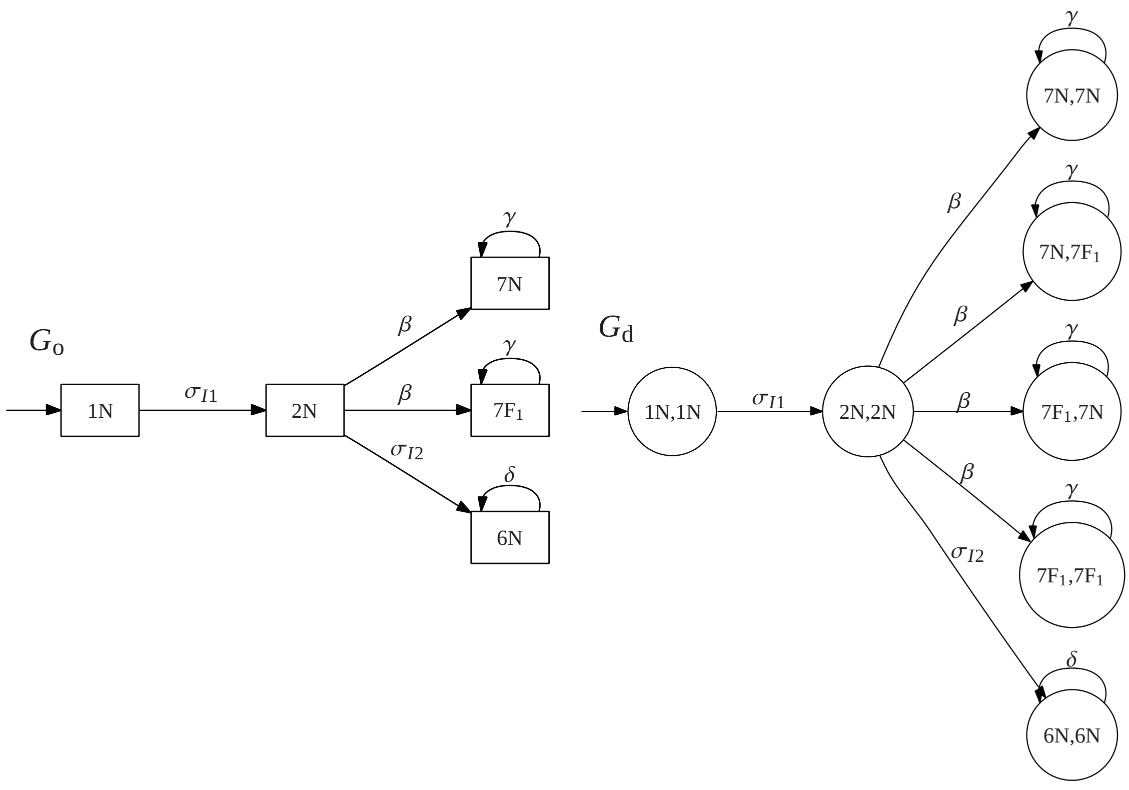
Figure 3.7: Automaton G for illustration of I-diagnosability.

3.4 Implementation in libFAUDES

The diagnosis methods and diagnoser computation as elaborated in the previous sections were integrated in the libFAUDES C++ software library [1] in the scope of this thesis. The library is based on the Standard Template Library (STL) and implements data structures and algorithms for finite automata and regular languages. Since it offers a powerful plug-in mechanism, the plug-in *diagnosis* was implemented for the integration of the discussed methods in the library.

3.4.1 Automata and Sets in libFAUDES

For better understanding of the diagnosis plug-in, we give a short introduction to the automata classes available in libFAUDES (see also Figure 3.10).



(a) G_o for failure type F_1. (b) G_d for failure type F_1.

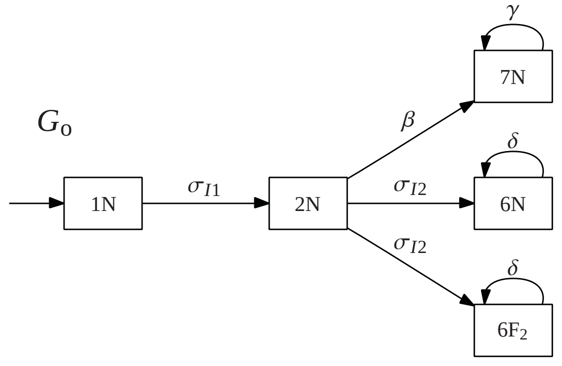
Figure 3.8: I-diagnosability test automata for failure type F_1.

Type is the base class for all libFAUDES objects and provides a uniform I/O interface which supports reading and writing of the object configuration of derived classes. It inherits *vGenerator* which is a virtual version of a plain generator with no attributes and is the base class for all generators.

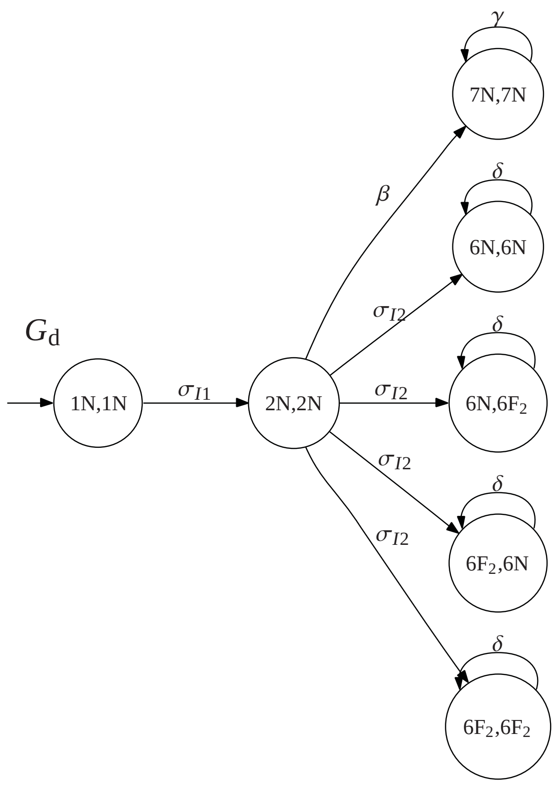
TaGenerator provides functions that allow read and write access to the core members events, states and transitions. It is a template class which is indicated by the template parameter T in its identifier. Template classes enable generic programming techniques in C++ and thus are an efficient way of defining and modifying properties of classes. *TaGenerator* implements template parameters to specify attribute classes for

- a global attribute of the generator *(class GlobalAttr)*,
- state attributes *(class StateAttr)*,
- event attributes *(class EventAttr)*,
- transition attributes *(class TransAttr)*.

The attributes itself are classes derived from *AttributeVoid*, *AttributeFlags*, or *AttributeCFlags*, which inherit from one another in that order. *AttributeVoid* is the minimal interface an attribute



(a) G_o for failure type F_2.



(b) G_d for failure type F_2.

Figure 3.9: I-diagnosability test automata for failure type F_2.

template parameter must provide and is the base class for all attribute implementations. *Attibute-Flags* provides additional semantics for boolean flags and *AttributeCFlags* moreover models event controllability and observability properties.

The *TcGenerator* inherits from the *TaGenerator* and additionally adds an interface for events with controllability and observability attributes, i. e., an event can now be controllable, observable or forcible. A plain finite state automaton with controllability properties can be modelled by using a *TcGenerator* with *AttributeCFlags* for the event attribute parameter and *AttributeVoid* for the other parameters. For convenience, this type is defined as *cGenerator*.

The libFAUDES library furthermore provides several container classes—among them the *TaIn-dexSet* which is a set of indices with attributes, and the *TaNameSet* which is a set of indices with symbolic names and attributes. For further information the reader is referred to [1].

3.4.2 Diagnoser Structure and Handling

The structure of the diagnoser is defined in the template class *faudes::TdiagGenerator*. As all the classes and functions of the diagnosis plug-in, *TdiagGenerator* is part of the namespace *faudes*.

Figure 3.10 shows the the inheritance diagram of the class *TdiagGenerator* that is defined as

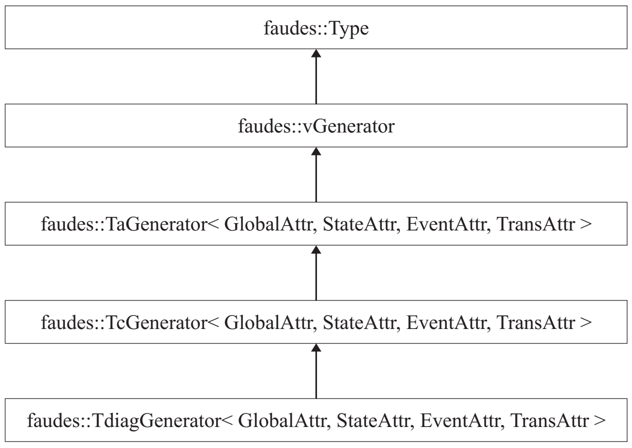

faudes::Type
faudes::vGenerator
faudes::TaGenerator< GlobalAttr, StateAttr, EventAttr, TransAttr >
faudes::TcGenerator< GlobalAttr, StateAttr, EventAttr, TransAttr >
faudes::TdiagGenerator< GlobalAttr, StateAttr, EventAttr, TransAttr >

Figure 3.10: Inheritance diagramm of class TdiagGenerator.

```
template <class GlobalAttr, class StateAttr, class EventAttr, class TransAttr>
class TdiagGenerator : public TcGenerator<GlobalAttr, StateAttr, EventAttr,
    TransAttr>.
```

As its base classes *TcGenerator* and *TaGenerator*, it is realized as a template class with the template parameters *GlobalAttr*, *StateAttr*, *EventAttr*, and *TransAttr*. For the standard concept of diagnosers we use the configuration illustrated in Figure 3.11.

In the following the remaining classes of the plug-in are introduced.

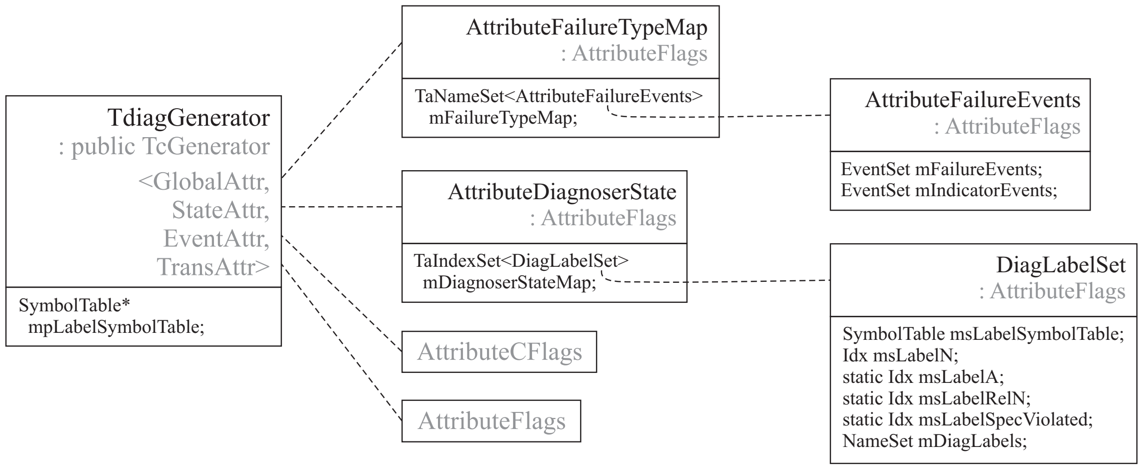


Figure 3.11: The class structure of a standard diagnoser including the member variables.

class AttributeFailureEvents

AttributeFailureEvents is derived from *AttributeFlags* and has got the two *EventSet* member variables *mFailureEvents* and *mIndicatorEvents*. They store the set of failure events Σ_{fi} and the corresponding set of indicator events Σ_{Ii} for a particular failure type.

class AttributeFailureTypeMap

The class *AttributeFailureTypeMap* unites the failure and indicator events for all failure types. It is derived from *AttributeFlags* and holds the member variable *mFailureTypeMap* which is of the type *TaNameSet<AttributeFailureEvents>*. The entries of this *NameSet* are failure type names with objects of class *AttributeFailureEvents* as attributes. Thus, *Attribute-FailureTypeMap* captures the complete failure partition and indicator map for a generator. It is used as the global attribute for our diagnoser.

class DiagLabelSet

This class is derived from *AttributeFlags* and provides methods to manipulate its major member variable *mDiagLabels*. This *NameSet* stores the names of the diagnoser state labels of a particular diagnoser state estimate. The handling is supported by additional members as a *SymbolTable* and several predefined static label names.

class AttributeDiagnoserState

AttributeDiagnoserState inherits from *AttributeFlags*. It stores a complete diagnoser state information in its member *mDiagnoserStateMap*. This *TaIndexSet<DiagLabelSet>* maps the indices of diagnoser state estimates to instances of *DiagLabelSets* that contain the corresponding state labels. By using *AttributeDiagnoserState* as state attribute in *TdiagGenerator* we assign the necessary information to every diagnoser state.

Compared to *TcGenerator*, there are no special requirements for events and transitions in the diagnoser. Thus, event attributes are provided by the libFAUDES class *AttributeCFlags* and transition attributes by *AttributeVoid*.

For convenience this presented configuration of a diagnoser is defined as *diagGenerator*:

```
typedef TdiagGenerator<AttributeFailureTypeMap, AttributeDiagnoserState,
    AttributeCFlags, AttributeVoid> diagGenerator.
```

Since the new class *TdiagGenerator* is derived from *TcGenerator*, it inherits its structure and methods (for documentation see [1, C++ API]). Additionally, *TdiagGenerator* provides methods to

- add failure types or the whole failure partition, respectively,
- query the failure type of a certain failure event,
- set or read the diagnoser state attributes.

Read and write access to the diagnoser is established with the standard libFAUDES token reader/writer and is executed by the member functions *Read()* and *Write()*, respectively. All the attribute classes have got the same read/write mechanism and thus can easily be written to and read from *gen* files. The following *gen* file is a curtailed version of the diagnoser shown in Figure 3.2b. It illustrates the labelling of state 10, which contains the state estimates $7F_1$ and $12AF_1$. The failure partition *FailureTypes* partitions the failure events in two sets, F_1 and F_2. Note, that there are no indicator events, because I-diagnosability is not considered in this example.

```
<Generator>
"Diagnoser"

<Alphabet>
...
</Alphabet>

<States>
...
10
<GenStateEstimates>
7
<DiagLabels>
"F1"
</DiagLabels>
12
<DiagLabels>
"A"                    "F1"
</DiagLabels>
...
</States>

<TransRel>
...
</TransRel>

<InitStates>
...
</InitStates>

<MarkedStates>
</MarkedStates>

<FailureTypes>
"F1"
<FailureEvents>
"sigma_f1"
</FailureEvents>
<IndicatorEvents>
</IndicatorEvents>
"F2"
```

```
<FailureEvents>
"sigma_f2"  "sigma_f2_dash"
</FailureEvents>
<IndicatorEvents>
</IndicatorEvents>
</FailureTypes>

</Generator>
```

Of course, the single elements of a diagnoser can also be written separately. E. g., this following output could be the failure partition of the example in Chapter 3.1.2.

```
<FailureTypes>
"F1"
<FailureEvents>
"sigma_f1"
</FailureEvents>
<IndicatorEvents>
"gamma"
</IndicatorEvents>
"F2"
<FailureEvents>
"sigma_f2"
</FailureEvents>
<IndicatorEvents>
"delta"
</IndicatorEvents>
"F3"
<FailureEvents>
"sigma_f3"
</FailureEvents>
<IndicatorEvents>
"delta"
</IndicatorEvents>
</FailureTypes>
```

Some other functions that do not belong directly to the diagnoser structure, but are still related to the diagnosis framework are stored in separate files within the *diagnosis plug-in*. The function

```
void ComputeDiagnoser(const cGenerator& G, const AttributeFailureTypeMap&
    rAttrFTMap, diagGenerator& G_diag);
```

requires a *cGenerator* and an *AttributeFailureTypeMap* as input and from there calculates the diagnoser according to Chapter 3.2.

3.4.3 Diagnosability Tests

This section describes the implementation of the diagnosability tests in libFAUDES. First the implementation of the diagnosability test as shown in Chapter 3.1.1 is described, and then the implementation of I-diagnosability as described in Chapter 3.1.2.

Testing for Diagnosability

The main function for testing standard diagnosability of a system G is

```
bool IsDiagnosable(const cGenerator& G, const AttributeFailureTypeMap&
    rFailureTypeMap, string& rReport).
```

It requires a generator, a failure type map and a report string as input parameters and returns the test result as a boolean value.

The function `IsDiagnosable()` then calls `MeetsDiagnosabilityAssumptions(const cGenerator& G, const AttributeFailureTypeMap& rFailureTypeMap, string& rReport)` to check whether

- all failure and indicator events are part of the generators alphabet,
- all failure events are unobservable,
- G is live,
- there do not exist cycles of unobservable events in G.

If all of these assumptions are met, the diagnosability test as described in Algorithm 3.1 is started. In order to have linear complexity in the number of failure types, the algorithm is applied to every single failure type separately:

1. First, `void ComputeGobs(const cGenerator& G, const string& rFailureType, const EventSet& rFailureEvents, diagGenerator& Go)` is called to compute generator G_o. Starting from the initial state with label N, `ComputeReachability()` determines the reachable states of the system G with exactly one observable transition. Is is done by means of a *depth-first search* (cp. [2]) that is aborted after the first observable event. The failure types which occur on these traces are tracked and the new state estimates and occurring failure types are stored as states in G_o. This reachability search is done for every new state in G_o until no further states in G are reachable.

2. Then, `void ComputeGd(const diagGenerator& Go, map<pair<Idx,Idx>,Idx>& rReverseCompositionMap, cGenerator& Gd)` computes G_d by evaluating the parallel composition of G_o with itself. Since G_d is implemented as a *cGenerator*, i. e., its states do not carry any labels, the mapping information of the states is stored in the `map<pair<Idx,Idx>,Idx> reverseCompositionMap`, which can be used for further manipulations.

3. Last, the function **bool** ExistsEvilCyclesInGd(**cGenerator**& G_d, **const diagGenerator**& G_o, **map**<**pair**<**Idx**,**Idx**>,**Idx**>& rReverseCompositionMap, **const string**& rFailureType, **string**& rReport) is called to check if there exist any cycles of states in G_d that correspond to states in G_o with mutually different failure labels. Therefore the function parses through the reverseCompositionMap and deletes all states in G_d that correspond to states in G_o with the same failure label. Then **bool** ExistsCycle(**const cGenerator**& G, **string**& rReport) checks if there exist any cycles in the remaining automaton. If so, the system G is stated not to be diagnosable.

If, for all failure types, there do not exist any offending cycles in G_d, the system is diagnosable and IsDiagnosable() returns *true*, otherwise it is not diagnosable and *false* is returned.

Testing for I-Diagnosability

Testing a system for I-diagnosability works pretty similar to the test of diagnosability. The core function to run the test is

```
bool IsIdiagnosable(const cGenerator& G, const AttributeFailureTypeMap&
    rFailureTypeMap, string& rReport).
```

It requires a generator, a failure type map (which contains both, failure and indicator events for every failure type) and a report string as input parameters and returns the test result as a boolean value.

As in the case of normal diagnosability the system has to fulfil the following assumptions which are checked by **bool** MeetsDiagnosabilityAssumptions(**const cGenerator**& G, **const** AttributeFailureTypeMap& rFailureTypeMap, **string**& rReport):

- all failure and indicator events have to be part of the generators alphabet,
- all failure events have to be unobservable,
- G has to be live,
- there should not exist any cycles of unobservable events in G.

If the system meets these requirements the test for I-diagnosability as described in Algorithm 3.2 is started. Since this algorithm is only defined for single failure types, we apply it to every failure type separately. The difference to the code structure shown before is that in the last step we only consider traces that start with an indicator event following a failure event. This is practically done by pruning G_d such that only those traces remain in the automaton, which can be viewed as an additional step before the last step. Then, as before, states with unequal failure labels are deleted and it is checked if there exist any cycles in the remaining graph.

Thus, the code to test for I-diagnosability for an individual failure type does the following:

1. The generator G_o is calculated by ComputeGobs().
2. ComputeGd() computes G_d.
3. Starting for the initial state, TrimNonIndicatorTracesOfGd() extracts all traces that start with an indicator event that follows a failure event. This is done by recursively deleting the transitions in every path of G_d until a transition with an indicator event that points to a state containing a failure label is found.
4. ExistsViolatingCyclesInGd () is called to check if there exist any cycles in G_d that have unequal failure labels. If so, the system G is stated not to be I-diagnosable.

If, for all failure types, there do not exist any offending cycles in G_d, the system is I-diagnosable and IsDiagnosable () returns *true*, otherwise it is not I-diagnosable and *false* is returned.

Chapter 4

Diagnosability with respect to a Specification

Beyond the diagnosability tests with respect to a failure partition that have been presented in the last chapter, we are also interested in a method to test a DES' diagnosability with respect to a specification language. Testing in the specification language framework is more flexible than testing with respect to failure types because now the complete behaviour of the system can be modelled as desired in a specification language. Every violation of a the specification is equivalent to the occurrence of a global failure. Qiu and Kumar [8] presented the notion of *codiagnosability* for systems with several local diagnosers. We will use their definition for a single diagnoser which we call *diagnosability with respect to a specification* and will provide an algorithm to test it.

Note that, in the specification language framework, I-diagnosability is not of interest any more because one can specify the behaviour of the system after each irregularity separately.

4.1 Definition and Testing Procedure

Given is a system G and its generated language $L(G)$. The specification language $K \subseteq L$ is generated by the specification automaton H, i. e., $L(H) = K$, where K does not need to be prefix-closed. Unlike before, in the specification framework G is allowed to contain deadlocking states and or cycles of unobservable events.

Definition 4.1 (Language Diagnosability). Given the observation mask $p_o : \Sigma^* \to \Sigma_o^*$, a system G is diagnosable with respect to the specification $K \subseteq L(G)$ if

$$(\exists n \in \mathbb{N})(\forall s \in L(G) - K)(\forall st \in L(G) - K, |t| \geq n \text{ or } st \text{ deadlocks})$$
$$\Rightarrow (\forall u \in p_o^{-1} p_o(st) \cap L(G),\ u \in L(G) - K).$$

This definition means the following: Given a string s in $L(G)$ that violates the specification K and another string t such that t is a sufficiently long extension of s in the "faulty language" or st is deadlocking. If every trace in $L(G)$ that is indistinguishable from st is part of the faulty language $L(G)-K$, the system G is said to be diagnosable with respect to the specification K.

Obviously, a string $s \in L(G)$ does not violate the specification if $s \in L(G) \| K$, but violates the specification if $s \in L(G)-K$. To verify diagnosability, we now have to check if there exists a string in $L(G)-K$ that is forever indistinguishable from a string in $L(G) \| K$. The following algorithm is based on [5] and describes a method to test a system G for diagnosability with respect to a given specification K.

Algorithm 4.1 (Diagnosability test with respect to a specification). For a given system $G = (X, \Sigma, \delta, x_0\, X_\mathrm{m})$ with an observation mask $p_\mathrm{o}\colon \Sigma^* \to \Sigma_\mathrm{o}^*$ do the following:

1. Construct an automaton $\tilde{G} = (\tilde{X}, \Sigma, \tilde{\delta}, x_0, \tilde{X}_m)$ with $L(\tilde{G}) = L(G)$ and $L_\mathrm{m}(\tilde{G}) = L(G)-K = L(G)\cap K^c$. Now every trace in $L(G)$ that violates the specification will lead to a marked state in $L(\tilde{G})$.

2. From $\tilde{G}$, construct a finite state automaton $G_\mathrm{o} = (X_\mathrm{o}, \Sigma_\mathrm{o}, \delta_\mathrm{o}, x_0^\mathrm{o}, X_\mathrm{m}^\mathrm{o})$ with language $L(G_\mathrm{o}) = p_\mathrm{o}(L(\tilde{G}))$ as follows:

   - $X_\mathrm{o} = \{(x, f) \mid x \in X_1 \cup \{x_\mathrm{o}\}, f \subseteq \{F\}\}$ is the finite set of states, where $X_1 = \{x \in X \mid \delta(x', \sigma) = x,$ with $x' \in X, p_\mathrm{o}(\sigma) \neq \varepsilon\}$ is the set of states in G that can be reached through an observable transition, and f is the label which indicates that a violation of the specification has occurred.
   - Σ_o, the set of observable events, is the set of events labels for G_o.
   - δ_o is the set of transitions. $\delta_\mathrm{o}((x, f), \sigma) = (x', f')$ if and only if there exists a path $(x, \sigma_1, x_1, \ldots, \sigma_n, x_n, \sigma, x')$, $n \geq 0$ in $\tilde{G}$ such that $\forall i \in \{1, 2, \ldots, n\}$, $p_\mathrm{o}(\sigma_i) = \varepsilon$, $p_\mathrm{o}(\sigma) = \sigma$, $f' = \emptyset$ if $x' \notin \tilde{X}_m$, $f' = F$ if $x' \in \tilde{X}_m$; otherwise $\delta_\mathrm{o}((x, f), \sigma)$ is not defined.
   - $x_0^\mathrm{o} = (x_0, \emptyset) \in X_\mathrm{o}$ is the initial state.
   - $X_\mathrm{m}^\mathrm{o} = \emptyset$.

3. Compute $G_\mathrm{d} = G_\mathrm{o} \| G_\mathrm{o} = (X_\mathrm{d}, \Sigma_\mathrm{o}, \delta_\mathrm{d}, x_0^\mathrm{d}, X_\mathrm{m}^\mathrm{o})$, where

   - $X_\mathrm{d} = \{(x_1^\mathrm{o}, x_2^\mathrm{o}) \mid x_1^\mathrm{o}, x_2^\mathrm{o} \in X_\mathrm{o} \times X_\mathrm{o}\}$ is the set of states.
   - Σ_o is the set of events labels for G_d.
   - δ_d is the set of transitions. $\delta_\mathrm{d}((x_1^\mathrm{o}, x_2^\mathrm{o}), \sigma) = (y_1^\mathrm{o}, y_2^\mathrm{o})$ if and only if $\delta_\mathrm{o}(x_1^\mathrm{o}, \sigma) = y_1^\mathrm{o}$ and $\delta_\mathrm{o}(x_2^\mathrm{o}, \sigma) = y_2^\mathrm{o}$; otherwise $\delta_\mathrm{d}((x_1^\mathrm{o}, x_2^\mathrm{o}), \sigma)$ is not defined.
   - $x_0^\mathrm{d} = (x_0^\mathrm{o}, x_0^\mathrm{o}) \in X_\mathrm{d}$ is the initial state.
   - $X_\mathrm{m}^\mathrm{o} = \emptyset$.

4. Check whether

   (a) there exists an unmarked state in $\tilde{G}$ with a trace of unobservable transitions that deadlocks in the marked states of $\tilde{G}$ or leads to a cycle of unobservable events in the marked states of $\tilde{G}$,

   (b) there exists a state $x = ((x^1, f^1), (x^2, f^2))$, $f^1 \neq f^2$, in G_d, such that in $\tilde{G}$, state x^1 or x^2 leads with an unobservable trace to a cycle of unobservable events in the marked states,

   (c) there exists a state $x = ((x^1, f^1), (x^2, f^2))$ in G_d, such that $f^1 \neq f^2$ and state (x^i, f^i), $f^i = F$, $i = 1, 2$ deadlocks in G_o,

   (d) there exists in G_d a cycle $cl = (x_1, \sigma_1, x_2, \ldots, x_n, \sigma_n, x_1)$, $n \geq 1$, $x_i = ((x_i^1, f_i^1), (x_i^2, f_i^2))$, $i = 1, 2, \ldots, n$, such that $f_1^1 \neq f_1^2$.

   If any of these exist, the system G is not diagnosable with respect to the specification K; otherwise it is diagnosable with respect to the specification K.

To illustrate the test according to the algorithm and the new conditions (a), (b) and (c) stated in step 4, consider the following examples. The events σ_f and σ_uo are unobservable, while the other events are observable.

Figure 4.1 depicts a system G and a specification automaton H with $L(H) = K \subseteq L(G)$. $\tilde{G}$ marks $L(G) - K$ which is the part of $L(G)$ that violates the specification K. G_o records every violation of the specification as long as there follows an observable event and labels it as a failure. The automaton G_d is the parallel composition of G_o with itself as before. Since there does not exist any cycles or deadlocking state as stated in the conditions of the last step in Algorithm 4.1, this system is diagnosable with respect to the specification K.

As stated in condition (a), diagnosability can be violated, if a system contains a unobservable trace that leads from an unmarked state into an unobservable loop and thereby violates the specification. In this case G_o and G_d do not capture the occurrence of the failure, but it can be identified in $\tilde{G}$: The example given in Figure 4.2 shows an unobservable transition with event σ_f from the unmarked state 3 in $\tilde{G}$ to an unobservable loop at state 4. This is not detectable for a diagnoser and hence this system is not diagnosable.

An example that violates condition (b) is considered in Figure 4.3. Here, there exists an unobservable loop in the marked states of $\tilde{G}$ which is reachable through an unobservable trace from the marked state 5. From G_d, we know that the specification violation has not yet been identified at state 5 and thus the system G is not diagnosable.

To illustrate condition (c), Figure 4.4 depicts the analysis of a system G which is not diagnosable because of a deadlocking marked state in G_o. The deadlocking states (3N,5F) and (5F,3N) in G_d

are evidence of two traces in G_{o}, where one trace is faulty and the other one is not. Since 5F is deadlocking in G_{o}, diagnosability is violated. Note, that in this example, the self-loop of the unobservable event σ_{uo} does not violate diagnosability as it is allowed by the specification.

4.2 Implementation in libFAUDES

To test a system for diagnosability with respect to a specification, we provide the function

```
bool IsDiagnosable(const cGenerator& G, const cGenerator& H, string& rReport)
```

which requires a generator G and a specification automaton H as input parameters and returns the test result as boolean. In case of a negative test result, the report string returns additional information about the failure condition.

IsDiagnosable() first marks all states in G and H. In order to make sure that transitions of events in G that do not occur in the specification do not violate diagnosability, self-loops of all those events are inserted in all states of H.

Next, $\tilde{G}$ with $L_{\mathrm{m}}(\tilde{G}) = L(G) - K = L(G) \cap K^c$ is evaluated by computing the parallel composition of G and the automaton returned by LanguageComplement(H). (Note that the parallel composition can be used here as equivalent of the language intersection because both automata have got the same alphabet.) Then ComputeGobs(**const cGenerator**& $\tilde{G}$, **diagGenerator**& G_{o}) generates G_{o} which is done in a similar manner as in Chapter 3.4.3 for diagnosability with respect to a failure partition. As it will be needed for the decentralized diagnosis (cp. Chapter 5.2) also nondeterministic automata with multiple initial states are allowed as input parameters and there exists a unique failure label F which is used for every violation of the specification. Next, G_{d} is calculated by ComputeGd(**const diagGenerator**& G_{o}, **map**<**pair**<**Idx**,**Idx**>,**Idx**>& rReverseCompositionMap, **cGenerator**& G_{d}) which computes the parallel composition of G_{o} with itself and stores the mapping information in the reverseCompositionMap, as before.

To determine diagnosability the conditions stated in step 4 of Algorithm 4.1 have to be checked:

1. **bool** ExistsViolatingCyclesInGd (**cGenerator**& G_{d}, **const diagGenerator**& G_{o}, **map**<**pair**<**Idx**,**Idx**>,**Idx**>& rReverseCompositionMap, **const string**& rFailureType, **string**& rReport) checks if there exist any cycles of states in G_{d} that correspond to states in G_{o} with mutually different failure labels. Therefore the function deletes all states in G_{d} and entries in rReverseCompositionMap that correspond to states in G_{o} with the same failure label—the remaining automaton of G_{d} is now called $G_{\mathrm{d,pruned}}$. **bool** ExistsCycle (**const cGenerator**& G, **string**& rReport) then checks if there exist any cycles in $G_{\mathrm{d,pruned}}$.

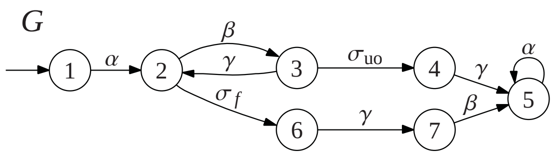
2. **bool** IsComplete(**const cGenerator**& G_{o}, **StateSet**& rDeadStates) extracts all deadlocking states from G_{o} and then it is checked whether one of these states carries a failure label in $G_{\mathrm{d,pruned}}$. This

is done by parsing through the pruned rReverseCompositionMap and reading the failure labels of the corresponding G_o states.

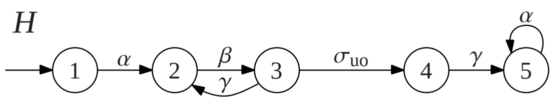
3. To check for unobservable deadlocks in the marked states of $\tilde{G}$ which result from an unobservable trace starting in the unmarked states, we first use **bool** IsComplete(**const cGenerator**& $\tilde{G}$, **StateSet**& rDeadStates) to extract all deadlocking states from $\tilde{G}$. Among these, we then extract the marked states and ComputeBackwardReachability(**const cGenerator**& $\tilde{G}$, **const Idx** state, **const EventSet**& rConsideredEvents, **StateSet**& rReachStates) finds all states that have unobservable traces leading to these marked deadlocks. Now we check if there exist any unmarked states among the start states of the unobservable traces.

4. To determine if there exists an unobservable cycle that violates diagnosability, we first extract all starting states of unobservable cycles in $\tilde{G}$. Therefore a copy is made, all observable transitions are deleted and CycleStartStates(**const cGenerator**& $\tilde{G}$, **StateSet**& unobsCycleOrigins) saves all starting states of cycles unobsCycleOrigins. Addtionally, ComputeBackwardReachability(**const cGenerator**& $\tilde{G}$, **const Idx** state, **const EventSet**& rConsideredEvents, **StateSet**& rReachStates) finds all states in $\tilde{G}$ that lead with unobservable traces to one of these states and adds them to unobsCycleOrigins. Now it has to be checked if any of the states in unobsCycleOrigins
   (a) is not marked in $\tilde{G}$,
   (b) forms part of a state label in $G_{\mathrm{d,pruned}}$. This is done by parsing though the prunded rReverseCompositionMap and evaluating the corresponding state labels of G_o.

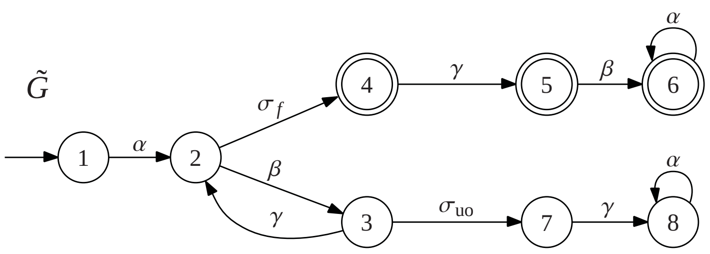
If any of these tests is positive, IsDiagnosable() returns *false*, otherwise *true*.



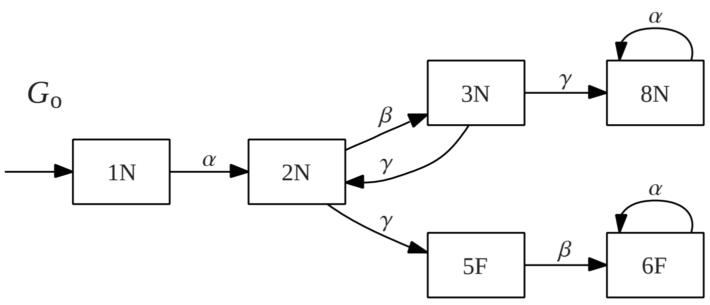
(a) The system G. [8]



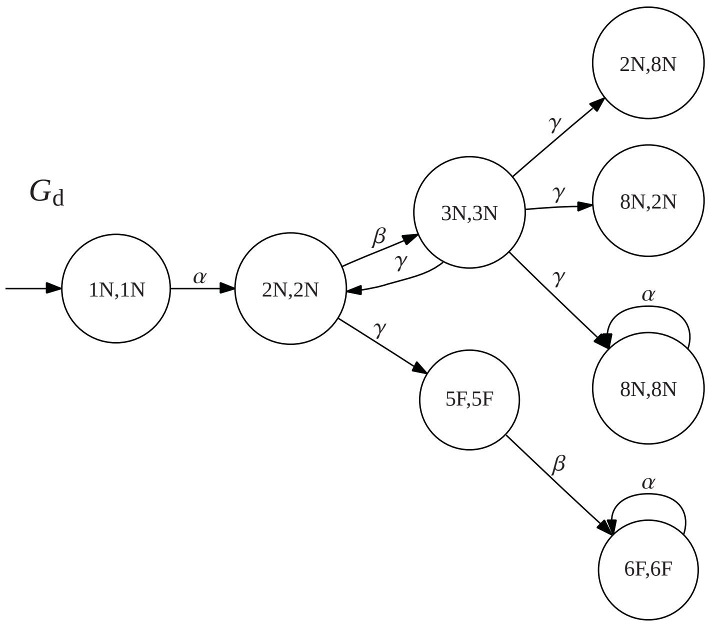
(b) The specification automaton H with $L(H) = K$. [8]



(c) The generator $\tilde{G}$ marks $L(G) - K$.

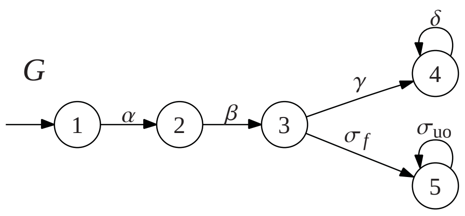


(d) The diagnoser G_{o} represents the observable behaviour of $\tilde{G}$ and implements every transition into a marked state as a failure F.

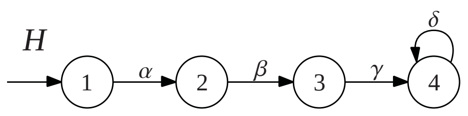


(e) The automaton $G_{\text{d}} = G_{\text{o}} \parallel G_{\text{o}}$.

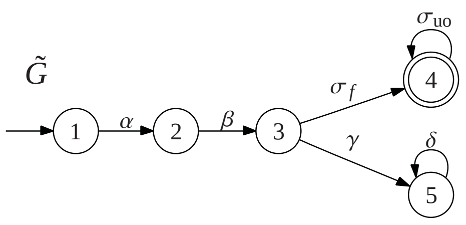
Figure 4.1: Diagnosability test automata for the system G with respect to the specification $K = L(H)$. G is diagnosable because none of the conditions of the last step of Algorithm 4.1 is fulfilled.



(a) The system G.

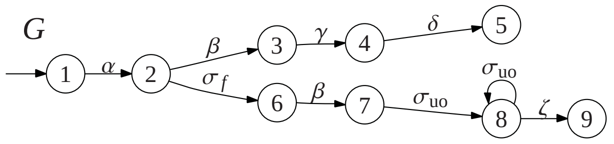


(b) The specification automaton H.

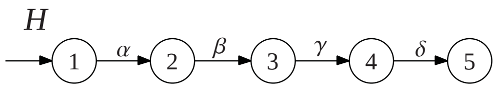


(c) The generator $\tilde{G}$.

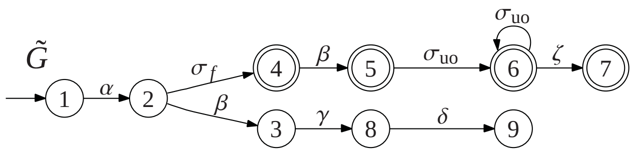
Figure 4.2: Test of diagnosability of system G with respect to $K = L(H)$. There exists an unobservable violation of the specification and thus G is not diagnosable.



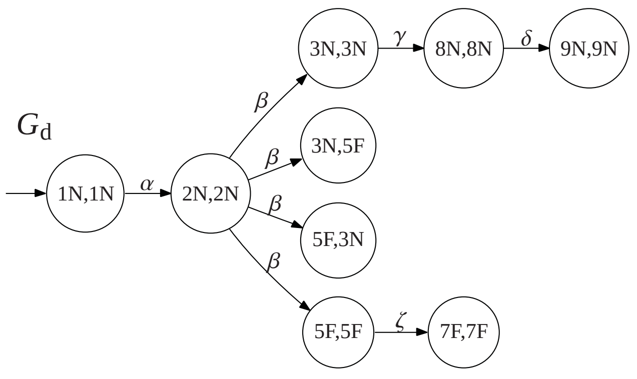
(a) The system G.



(b) The specification automaton H.

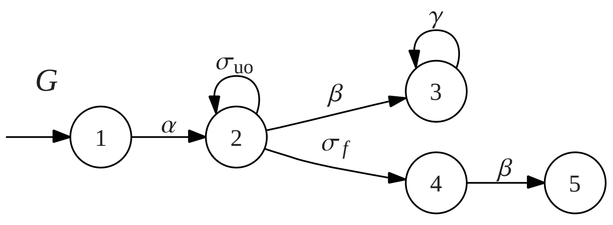


(c) The generator $\tilde{G}$.

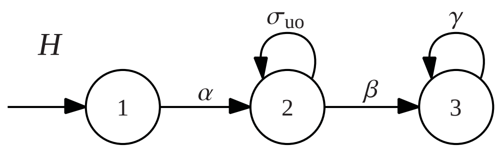


(d) The automaton G_d.

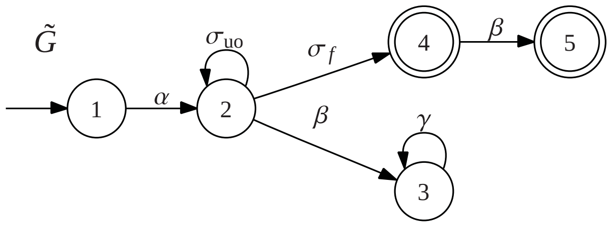
Figure 4.3: Test of diagnosability of system G with respect to $K = L(H)$. There exists an unobservable cycle in the marked states of $\tilde{G}$ that has an unobservable origin at state 5. Since state 5 is ambiguous in G_d (cp. states (3N,5F) and (5F,3N)), G is not diagnosable with respect to K.



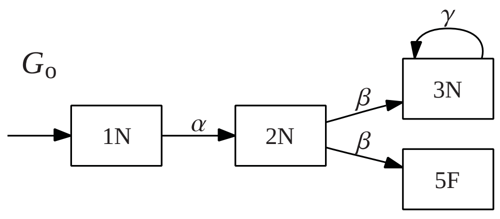
(a) The system G.



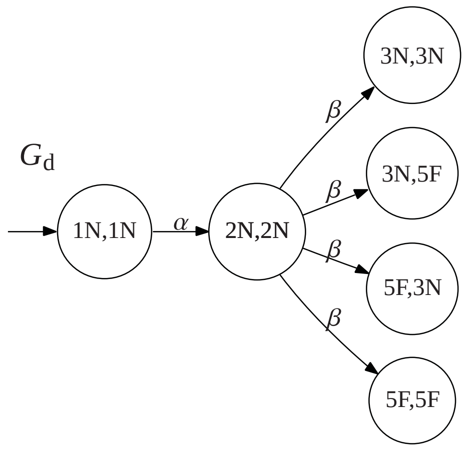
(b) The specification automaton H.



(c) The generator $\tilde{G}$.



(d) The diagnoser G_{o}.



(e) The automaton G_{d}.

Figure 4.4: Diagnosability test of the System G. There exists a deadlock at state 5 in G_{o}. From G_{d}, the system status is uncertain at the deadlock because of the ambiguous states (3N,5F) and (5F,3N). Hence, the system G is not diagnosable with respect to K.

Chapter 5

Decentralized Diagnosis

So far, we looked at methods to determine diagnosability for an entire system with an entire failure partition or an entire specification. For large-scale plants, these approaches have the drawback that their algorithms become computationally infeasible. Therefore, it stands to reason to extend the notion of diagnosability for modular plants and plants divided into subsystems, respectively.

In this chapter we will propose *decentralized diagnosis for local specifications* as a method for an abstraction based failure diagnosis of DES.

5.1 Related Work

In [8], Qiu *et al.* presented the notion of *codiagnosability*. It applies to a global representation of the plant G which is observed by m local diagnosers with respect to a global specification K. Each local diagnoser uses its own observations of the system and no communication between the diagnosers is needed. Codiagnosability requires that any violation of the specification is detectable within a bounded number of transitions of G by at least one local diagnoser. The authors also introduce codiagnosability in the failure event framework and present a polynomial algorithm to test for codiagnosability. Furthermore they extend their notion to *strong codiagnosability* which additionally requires the detection of non-faulty traces within a bounded number of transitions.

Zhou *et al.* [17] focus on a modular plant observed with regards to a global specification. They introduce the notion of *modular diagnosability* that allows to diagnose every failure using a set of local diagnosers. The computation of these diagnosers only depends on the local subplants and thus does not require the construction of the global plant model. A system is tested for modular diagnosability by reducing it to codiagnosability, hence the construction of the overall plant model is needed for testing procedure.

5.2 Decentralized Diagnosis for Modular Plant and Specification

Consider a modular plant that consists of m subplants $G_i = (X_i, \Sigma_i, \delta_i, x_{0,i})$, $i \in \mathcal{I} := \{1, \ldots, m\}$ over the alphabets Σ_i such that the overall plant $G = (X, \Sigma, \delta, x_0)$ is defined as $G := \|_{i\in\mathcal{I}} G_i$. We have m locally diagnosability specifications $K_i \subseteq \Sigma_i^*$, $i \in \mathcal{I}$ such that the overall diagnosability specification evaluates to $K := \|_{i\in\mathcal{I}} K_i$.

We now want to verify diagnosability of G with respect to K without having to evaluate G and K.

5.2.1 Decentralized Diagnosability for Individual Subsystem

We present a method that allows to conclude diagnosability for G from a local computation. In summary, we evaluate an abstraction of the local plant that incorporates the behaviour of the other subsystems. In this abstraction, we replace all transitions of events that do not form part of the local alphabet by ε-transitions and compute the parallel compositions of the original local plant with the abstraction. Hereby, we get another version of the local plant (with a usually bigger state space) that on its part also captures the behaviour of the other subsystems. With a generalized version of the language diagnosability notion, we are then able to verify decentralized diagnosability for the local plant and its local specification.

To begin with, define $\Sigma_\cap := \bigcup_{i,j\in\mathcal{I}, i\neq j}(\Sigma_i \cap \Sigma_j)$ as the *set of shared events* and the corresponding *local sets of shared events* $\Sigma_{i,\cap} = \Sigma_\cap \cap \Sigma_i$, $i \in \mathcal{I}$. We define a version of the L-observer condition in [15].

Definition 5.1 (Loop-preserving Observer). Let $p\colon \Sigma^* \to \hat{\Sigma}^*$ be the natural projection for $\hat{\Sigma} \subseteq \Sigma$ and let G be an automaton. Then p is a loop-preserving observer for $L(G)$ with the bound N if for all $s \in L(G)$ and $t \in \hat{\Sigma}^*$

$$p(s)t \in p(L(G)) \Rightarrow \exists u \in \Sigma^* \text{ such that } su \in L(G) \text{ and } p(su) = p(s)t$$

and for all such u, $|u| < N|t|$.

This definition states the following: Given a trace s generated by the system and a trace t within the abstraction language. Then, the natural projection p is a loop-free observer with bound N if for all extensions of the abstracted trace s in the abstraction language it holds that there also exists an extension u in the system's language such that in the abstraction, su projects onto the extended string in the abstraction. Additionally, u is bounded by $|u| < N|t|$. Hence, the extension u in the system's language cannot be arbitrarily long and it follows, that every loop in the system also has to appear in the abstraction.

Now, we select one particular plant G_j, $j \in \mathcal{I}$ and write $\mathcal{I}_j := \mathcal{I} - \{j\}$ and $G_{\mathcal{I}_j} := \|_{i=1,i\neq j}^{m} G_i$ over the alphabet $\Sigma_{\mathcal{I}_j} := \bigcup_{i\in\mathcal{I}_j} \Sigma_i$ for the composition of the remaining components. Then, we define the alphabets $\hat{\Sigma}_i \supseteq \Sigma_{i,\cap}$ that are as small as possible but such that the natural projections $p_i\colon \Sigma_i^* \to \hat{\Sigma}_i^*$

are loop-preserving observers for the local plants G_i, $i \in \mathcal{I}_j$. The overall abstraction alphabet is $\hat{\Sigma}_{\mathcal{I}_j} := \bigcup_{i \in \mathcal{I}_j} \hat{\Sigma}_i$, the overall natural projection is $p_{\mathcal{I}_j}\colon \Sigma^*_{\mathcal{I}_j} \to \hat{\Sigma}^*_{\mathcal{I}_j}$. The abstraction $\hat{G}_{\mathcal{I}_j}$ over $\hat{\Sigma}_{\mathcal{I}_j}$ of the remaining plant $G_{\mathcal{I}_j}$ is computed by evaluating

$$L(\hat{G}_{\mathcal{I}_j}) = p(L(G_{\mathcal{I}_j})) = ||_{i \in \mathcal{I}_j} p_i(L(G_i)).$$

We use loop-preserving observers for the abstraction of the local plants G_i, $i \in \mathcal{I}_j$, in order to ensure that the system's local loops that could violate diagnosability by executing arbitrarily long strings are incorporated in the abstraction.

It holds that $p_{\mathcal{I}_j}$ is a loop-preserving observer if all p_i, $i \in \mathcal{I}_j$ are loop-preserving observers.

Lemma 5.1 (Loop-preserving Observer). For $i \in \mathcal{I}_j$, let G_i be automata over the alphabet Σ_i and define the natural projections $p_i\colon \Sigma^*_i \to \hat{\Sigma}^*_i$ for $\hat{\Sigma}_i \subseteq \Sigma_i$. Also let $G_{\mathcal{I}_j}$, $\Sigma_{\mathcal{I}_j}$ and $p_{\mathcal{I}_j}$ be defined as above. Then $p_{\mathcal{I}_j}$ is a loop-preserving observer for $G_{\mathcal{I}_j}$ with the bound $N_{\mathcal{I}_j} := \sum_{i \in \mathcal{I}_j} N_i$ if p_i is a loop-preserving observer for G_i with the bound N_i for $i \in \mathcal{I}_j$.

Proof. Assume that p_i is a loop-preserving observer for G_i for $i \in \mathcal{I}_j$ and let $s \in L(G_{\mathcal{I}_j})$, $t \in \hat{\Sigma}^*_{\mathcal{I}_j}$ such that $p(s)t \in p(L(G_{\mathcal{I}_j}))$. It has to be shown that there is $u \in \Sigma^*$ such that $su \in L(G_{\mathcal{I}_j})$ and $p(su) = p(s)t$, and that for all such u, $|u| < N_{\mathcal{I}_j}|t|$.

Since $s \in L(G_{\mathcal{I}_j})$, $s_i := \theta_i(s) \in L(G_i)$ for $i \in \mathcal{I}_j$. Similarly, with $t_i := \hat{\theta}_i(t)$, $p_i(s_i)t_i \in p_i(L(G_i))$. Hence, for all i, there is a $u_i \in \Sigma^*_i$ such that $s_i u_i \in L(G_i)$ and $p_i(s_i u_i) = p_i(s_i)t_i$. Then, $||_{i \in \mathcal{I}_j} u_i \neq \emptyset$ and there is $u \in ||_{i \in \mathcal{I}_j} u_i$ such that $p(u) = t$. It remains to show that for all such u, $|u| < N_{\mathcal{I}_j}|t|$. By assumption, we know that for all i, $|u_i| < N_i|t_i|$. Furthermore, $u \in ||_{i \in \mathcal{I}_j} u_i$ implies that $|u| \le \sum_{i \in \mathcal{I}_j} |u_i|$. Hence, $|u| < \sum_{i \in \mathcal{I}_j} N_i|t_i| \le \sum_{i \in \mathcal{I}_j} N_i|t| = N_{\mathcal{I}_j}|t|$. ■

We now derive a method to find a substitute for a chosen subplant G_j that incorporates the behaviour of the other subplants. Therefore, a the nondeterministic automaton $H_j = (Q_i, \Sigma_i \cup \{\varepsilon\}, \nu_i, Q_{0,i}, X_{\mathrm{m},i})$ is computed for the chosen subplant G_j by the following procedure.

Algorithm 5.1 (Computation of H_j).

1. Find an alphabet $\hat{\Sigma}_j \supseteq \Sigma_{j,\cap}$ which is as small as possible but such that $p_j\colon \Sigma^*_j \to \hat{\Sigma}^*_j$ is an L-observer (loop-preserving observer is not required).

2. Compute $\hat{G}_j$ such that $L(\hat{G}_j) = p_j(L(G_j))$ and the overall abstraction $\hat{G} := \hat{G}_j \,||\, \hat{G}_{\mathcal{I}_j}$.

3. Determine the local view of $\hat{G}$ for the local plant G_j. Therefore, compute the automaton $\hat{H}_j$ from $\hat{G}$ by replacing all transitions in $\hat{G}$ with events that are not in $\hat{\Sigma}_j$ by ε-transitions. Hence, the transition structure and the state space of $\hat{H}_j$ is the same as for $\hat{G}$. However, $\hat{H}_j$ can be nondeterministic due to the ε-transitions.

4. Define $H_j := G_j \,||\, \hat{H}_j$. Hence, using $\hat{H}_j$, the behaviour of the other local plants is incorporated in H_j.

We suggest to use H_j and the local specification K_j to perform the diagnosability check. Since H_j is nondeterministic, we propose a generalization of the diagnosability condition. Our formulation holds for a general nondeterministic automaton G. It incorporates the fact that multiple states with different futures can be reached after each string in $L(G)$. Then, in order to be diagnosable, all possible futures in the nondeterministic automaton G have to fulfil diagnosability. Note that this condition reduces to Definition 4.1 if the automaton G is deterministic.

Definition 5.2 (Generalized Language Diagnosability). Define the natural projection $p_\mathrm{o}\colon \Sigma^* \to \Sigma_\mathrm{o}^*$ for $\Sigma_\mathrm{o} \subseteq \Sigma$. G is generalized diagnosable with respect to K if

$$\begin{aligned}&(\exists n \in \mathbb{N})(\forall s \in L(G) - K)(\forall x \in \delta(x_0, s))(\forall t \text{ such that } \delta(x,t)!,\ |t| \geq n \text{ or } st \text{ deadlocks})\\ &\Rightarrow (\forall u \in p_\mathrm{o}^{-1} p_\mathrm{o}(st) \text{ such that } \delta(x_0,u)! \text{ for some } x_0 \in X_0,\ u \notin K)\end{aligned}$$

Using H_j and Definition 5.2, we now provide a sufficient condition for decentralized diagnosability for an individual subsystem.

Proposition 5.1 (Decentralized Diagnosability for individual subsystem). Assume H_j is constructed as described above. It holds that G_j is diagnosable with respect to $\hat{K}_j := K_j \,||\, L(G)$ if

(a) H_j is generalized diagnosable with respect to K_j and

(b) every cycle in $\hat{G}$ contains at least one event in $\hat{\Sigma}_j$.

Proof. First, we introduce several natural projections needed in the proof: $p_i\colon \Sigma_i^* \to \hat{\Sigma}_i^*$, $\theta_i\colon \Sigma^* \to \Sigma_i^*$, $\hat{\theta}_i\colon \hat{\Sigma}^* \to \hat{\Sigma}_i^*$, $p_{\mathrm{o},i}\colon \Sigma_i^* \to \Sigma_{\mathrm{o},i}^*$, $\theta_{\mathrm{o},i}\colon \Sigma_\mathrm{o}^* \to \Sigma_{\mathrm{o},i}^*$, $p\colon \Sigma^* \to \hat{\Sigma}^*$ with $\hat{\Sigma} := \Sigma_j \cup \hat{\Sigma}_{\mathcal{I}_j}$. Here, $\Sigma_{\mathrm{o},j} := \Sigma_j \cap \Sigma_\mathrm{o}$ is the local set of observable events.

We assume that K_j is generalized diagnosable with respect to $H_j = G_j \,||\, \hat{H}_j$ and want to show that G is diagnosable with respect to $\hat{K}_j$. Let $s \in L(G) - \hat{K}_j$ and choose $\hat{n}_j := n_j \cdot |X|$, where n_j is taken from the generalized diagnosability test of K_j and H_j and $|X|$ is the state count of G. Now assume that $st \in L(G)$ and $|t| > \hat{n}_j$ or st deadlocks but $\exists u \in p_\mathrm{o}^{-1} p_\mathrm{o}(st) \cap L(G)$ such that $u \in \hat{K}_j$, i. e., G is not diagnosable with respect to $\hat{K}_j$.

If $|t| > \hat{n}_j$, since $\hat{n}_j = n_j \cdot |X|$, t passes at least n_j cycles in G. Then it must hold that at least one event in Σ_j must occur in each such cycle. Assume the contrary. Then, the cycle only contains events in $\Sigma - \Sigma_j$. Then, for all $i \in \mathcal{I}_j$ such that the cycle contains events in Σ_i, it holds that there are corresponding cycles in G_i. Since all p_i for $i \in \mathcal{I}_j$ are loop-preserving observers, the respective cycles must also appear in the associated $\hat{G}_i$. But then, a corresponding cycle with events only in $\hat{\Sigma}_{\mathcal{I}_j} - \hat{\Sigma}_j$ must exist in $\hat{G}_{\mathcal{I}_j}$, which also implies the existence of a cycle with events only in $\hat{\Sigma} - \hat{\Sigma}_j$ in $\hat{G}$. But this contradicts the assumption that each cycle in $\hat{G}$ must contain at least one event in $\hat{\Sigma}_j$. Hence, each cycle in G must contain events in Σ_j.

Since t passes at least n_j cycles in G, this implies that $t_j := \theta_j(t)$ contains at least n_j events in Σ_j, i. e., $|t_j| \geq n_j$. Furthermore, it follows from $\hat{t} := p(t) \in \hat{G}$ and $p_j(t_j) = \hat{\theta}_j(\hat{t})$, and the construction of $\hat{H}_j$ from $\hat{G}$ that $t_j \in H_j = G_j \parallel \hat{H}_j$. In addition, it holds that $s_j := \theta_j(s) \in L(H_j) - K_j$, and $u \in p_o^{-1} p_o(st) \cap L(G)$, while $u \in \hat{K}_j$ implies that $u_j := \theta_j(u) \in p_{o,j}^{-1} p_{o,j}(s_j t_j) \cap L(H_j)$ (concluded from $u \in \theta_j^{-1} p_{o,j}^{-1} p_{o,j} \theta_j(u) = \theta_j^{-1} p_{o,j}^{-1} \theta_{o,j} p_o(u) = \theta_j^{-1} p_{o,j}^{-1} \theta_{o,j} p_o(st) = \theta_j^{-1} p_{o,j}^{-1} p_{o,j} \theta_j(st)$) and $u_j \in K_j$. Now let $q \in \nu_j(q_{o,j}, s_j)$ such that $q_{t_j} \in \nu_j(q, t_j)$ exists (both states exist since $s_j t_j \in L(H_j)$). Hence, for $|t_j| > n_j$, we have a $q \in \nu_j(q_{o,j}, s_j)$ such that $\nu_j(q, t_j)!$ but there is a $u_j \in p_{o,j}^{-1} p_{o,j}(s_j t_j)$ such that $\nu_j(q_{o,j}, u_j)!$ and $u_j \in K_j$, which contradicts that H_j is generalized diagnosable with respect to K_j.

If st deadlocks in G, it can be shown that also $\hat{s}\hat{t} := p(s)p(t)$ deadlocks in $\hat{G}$. Assume that there is $\sigma \in \hat{\Sigma}$ such that $\hat{s}\hat{t}\sigma \in L(\hat{G})$. Then, there must be a $v \in \Sigma^*$ such that $stv\sigma \in L(G)$ because p is an $L(G)$-observer, which contradicts that st deadlocks in G. Define $\hat{x} := \hat{\delta}(\hat{x}_0, \hat{s}\hat{t})$ as the corresponding deadlock state in $\hat{G}$. Furthermore, since $st \in \theta_j^{-1}(s_j t_j)$ deadlocks in G, it must hold that either (i) $\nexists \sigma \in \Sigma_j - \hat{\Sigma}_j$ such that $s_j t_j \sigma \in L(G_j)$ or (ii) for all $\sigma \in \hat{\Sigma}_j$ such that $s_j t_j \sigma \in L(G_j)$, there is a $k \neq j$ with $\sigma \in \Sigma_k$ but $s_k \sigma \notin L(G_k)$. We denote the corresponding state in G_j as $x_j := \delta_j(x_{0,j}, s_j t_j)$. We now consider the parallel composition of G_j and $\hat{H}_j$, while respecting the construction of $\hat{H}_j$ from $\hat{G}$. It is readily observed that the state $(x_j, \hat{x})$ is reachable in $H_j = G_j \parallel \hat{H}_j$, as $\hat{\theta}_j(\hat{s}\hat{t}) = p_j(s_j t_j)$. In addition, $(x_j, \hat{x})$ is a deadlock state in H_j since neither further events in $\Sigma_j - \hat{\Sigma}_j$ (case (i)) nor further events in $\hat{\Sigma}_j$ (case (ii)) are possible. From above, we recall that at the same time $u_j := \theta_j(u) \in p_{o,j}^{-1} p_{o,j}(s_j t_j) \cap L(H_j)$ and $u_j \in K_j$. Now, there are two cases. If $|t_j| > n_j$, the above discussion for $|t| > \hat{n}_j$ shows that diagnosability of $\hat{K}_j$ for G holds. Otherwise, we choose $q_j \in \nu_j(q_{o,j}, s_j)$ such that $(x_j, \hat{x}) \in \nu_j(q_j, t_j)$. Hence, $q_j \in \nu_j(q_{o,j}, s_j)$ and $(x_j, \hat{x}) \in \nu_j(q_{o,j}, s_j t_j)$ deadlocks in H_j, while $u_j \in p_{o,j}^{-1} p_{o,j}(s_j t_j)$ such that $\nu_j(q_{o,j}, u_j)!$ and $u_j \in K_j$. Again, this contradicts that H_j is generalized diagnosable with respect to K_j.

Together, it holds that G must be diagnosable with respect to $\hat{K}_j$. ∎

5.2.2 Decentralized Diagnosability for the Overall System

Combining decentralized diagnosability for all local plants G_i and specifications K_i, we can state a sufficient condition for decentralized diagnosability for $K := \parallel_{i=1}^m K_i$ and $G := \parallel_{i=1}^m G_i$.

Theorem 5.1 (Decentralized Diagnosability). Assume that $K := \parallel_{i=1}^m K_i$ and $G := \parallel_{i=1}^m G_i$ are given as above. Then G is diagnosable with respect to K if Proposition 5.1 holds for all $i \in \mathcal{I}$.

Proof. Let $s \in L(G) - \overline{K}$ and choose $n := \max_i \hat{n}_i$, where $\hat{n}_i$ is taken from the proof of Proposition 5.1. Now assume that $t \in \Sigma^*$ such that $st \in L(G)$ and either (i) $|t| > n$ or (ii) st deadlocks. Also let $u \in p_o^{-1} p_o(st) \cap L(G)$. To prove diagnosability, we have to show that $u \in L(G) - \overline{K}$.

First observe that $s \in L(G) - \overline{K} = \parallel_{i=1}^m L(G_i) - \parallel_{i=1}^m \overline{K}_i$ implies that for some j, $s_j \notin \overline{\hat{K}}_j = \overline{K}_j \parallel L(G)$.

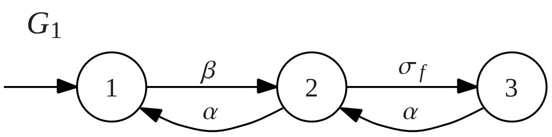
Now assume that (i) holds, i. e., $st \in L(G)$ and $|t| > n$ but there is $u \in p_o^{-1} p_o(st) \cap L(G) \cap \overline{K}$, i. e., G is not diagnosable with respect to K. Then, $u \in p_o^{-1} p_o(st) \cap L(G) \cap \theta_j^{-1}(\overline{K}_j) = p_o^{-1} p_o(st) \cap \overline{\hat{K}}_j$, while $|t| > n \geq \hat{n}_j$. But this means that G is not diagnosable with respect to $\overline{\hat{K}}_j$, which contradicts the assumption that Proposition 5.1 holds for j.

Finally assume that (ii) holds, i. e., $st \in L(G)$ deadlocks but there is $u \in p_o^{-1} p_o(st) \cap L(G) \cap \overline{K}$, i. e., G is not diagnosable with respect to K. Again, $u \in p_o^{-1} p_o(st) \cap L(G) \cap \theta_j^{-1}(\overline{K}_j) = p_o^{-1} p_o(st) \cap \overline{\hat{K}}_j$, while st deadlocks. But this means that G is not diagnosable with respect to $\hat{K}_j$, which contradicts the assumption that Proposition 5.1 holds for j. ■

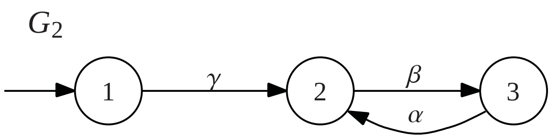
5.2.3 Illustration of the Conditions in the Decentralized Diagnosability Test

We now provide several examples to illustrate the relevance of the conditions in Algorithm 5.1 and Proposition 5.1 within our method to validate a system's decentralized diagnosability.

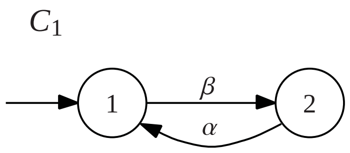
In the following, we call a subsystem G_i *locally diagnosable* if it holds that G_i is diagnosable with respect to its local specification K_i according to Definition 4.1. The specification K is generated by an automaton denoted as C, i. e., $L(C) = K$. Additionally, all events are assumed to be observable, except σ_f which is unobservable.



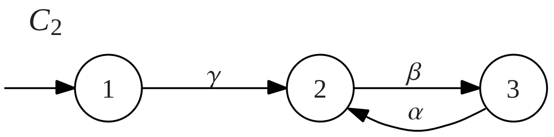
(a) The local subsystem G_1.



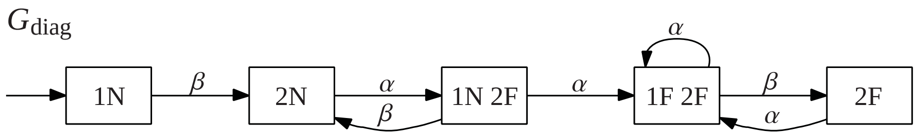
(b) The local subsystem G_2.



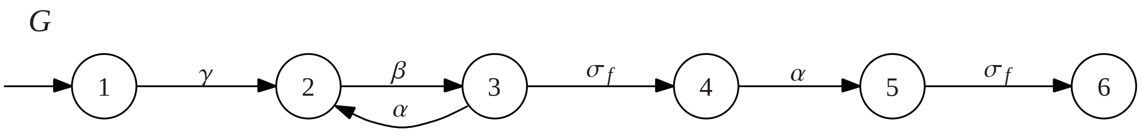
(c) The local specification automaton C_1.



(d) The local specification automaton C_2.



(e) The diagnoser G_{diag} of G_1.



(f) The overall system $G = G_1 \parallel G_2$.

Figure 5.1: Even though G_1 and G_2 are locally diagnosable, decentralized diagnosability does not hold for $G = G_1 \parallel G_2$ because G_2 inhibits the detection of the specification violation of G_1.

Firstly, we show that not including the behaviour of the other subsystems by the abstraction procedure can lead to a wrong test result. In other words, local diagnosability of all subsystems is not sufficient for decentralized diagnosability of the overall system in our approach. Consider a system G consisting of two subsystems G_1 and G_2 as depicted in Figure 5.1a and (b). The specifications K_1 and K_2 are chosen as given in Figure 5.1c and (d). For simplicity, $K_2 = L(G_2)$ so that local diagnosability trivially holds for G_2. From Definition 4.1 we know that G_1 is locally diagnosable with respect to K_1. A violation of K_1 is identified as soon as α occurs twice in a row (cp. G_{diag} in Figure 5.1e). However, from the overall system behaviour modelled in Figure 5.1f we can see that G_2 inhibits the repeated execution of α after the occurrence of σ_f. Hence, decentralized diagnosability does not hold for the overall system G.

Secondly, we illustrate the necessity that the abstractions of the subsystems are L-observers. Therefore, consider the system G consisting of the subsystems G_1 and G_2 as shown in Figure 5.2. Decentralized diagnosability works for G_1 with respect to K_1 and G_2 is assumed to be trivially diagnosable. For G_1, the local set of shared events is $\Sigma_{1,\cap} = \{\alpha, \gamma\}$ and gets extended to $\hat{\Sigma}_1 = \{\alpha, \gamma, \zeta, \sigma_f\}$ so that $p_1 : \Sigma_1^* \to \hat{\Sigma}_1^*$ is an L-observer. For G_2, the abstraction alphabet is chosen as $\hat{\Sigma}_2 = \{\alpha, \gamma\}$ and therewith $p_2 : \Sigma_2^* \to \hat{\Sigma}_2^*$ is *not* an L-observer. Hence, $\hat{G}$, $\hat{H}_1$, H_1 do not incorporate δ-transitions and the overall system $G = G_1 \parallel G_2$ is wrongly stated to be decentralized diagnosable. But from the overall system behaviour (Figure 5.2g), one can see that the deadlocking δ-transitions inhibits the detection of σ_f since γ cannot occur any more. Thus, the system G is not diagnosable because the abstraction of G is not an L-observer.

Thirdly, the system G depicted in Figure 5.3 shows the necessity of a loop-preserving abstraction. Its subsystems G_1 and G_2 have got the local specifications K_1 and K_2. G_1 is locally diagnosable with respect to K_1 and K_2 can be chosen freely so that G_2 is locally diagnosable as well. The abstraction alphabet of G_2 is chosen $\hat{\Sigma}_2 = \{\beta\}$, thus $p_2 : \Sigma_2^* \to \hat{\Sigma}_2^*$ is not a loop-preserving observer (i. e., there exists a local loop in G_2 which is not observed by the abstraction). The diagnosability test procedure states that this system is diagnosable, even though from the overall system behaviour it is obvious that the system is not diagnosable because of the self-loop of γ at state 5 which is not incorporated in $\hat{G}_2$.

Fourthly, we illustrate the need of condition (b) in Proposition 5.1 which requires every cycle in $\hat{G}$ to contain at least one event in $\hat{\Sigma}_j$. Figure 5.4 depicts a modular system G consisting of two subsystems G_1 and G_2, where G_2 is assumed to be decentralized diagnosable (e. g., by is choosing $K_2 = L(G_2)$). We now look at the test of decentralized diagnosability for G_1: The abstraction alphabets are $\hat{\Sigma}_1 = \{\zeta\}$ and $\hat{\Sigma}_2 = \{\beta, \zeta\}$ such that the abstractions result to $\hat{G}_1$ and $\hat{G}_2$, where p_1 is an L-observer and p_2 is a loop-preserving observer. $\hat{G}$ and H_1 result from Algorithm 5.1 and H_1 is generalized diagnosable with respect to K_1. However, $\hat{G}$ contains a β self-loop and $\beta \notin \hat{\Sigma}_1$. From the overall system behaviour (h) one can see that the cycle containing β at states 7 and 8 inhibits diagnosability after the occurrence of σ_f.

5.2.4 Computational Complexity

With regards to the computational complexity the test for decentralized diagnosability requires the computation of H_j which needs

- computation of observers for all $i \in \mathcal{I}$ (polynomial),
- parallel composition to obtain $\hat{G}$ (product state space of the abstractions, usually much smaller than original automata),
- replacement by ε-transitions (linear in number of transitions of $\hat{G}$),
- check if cycles in $\hat{G}$ without events in $\hat{\Sigma}_i$ exist (linear in number of transitions and states),
- parallel composition $G_j \parallel \hat{H}_j$ (product state space of $\hat{G}$ and G_j),

and diagnosability test for H_j with respect to K_j as proposed in [7] (see also Chapter 4) which is polynomial as well.

5.3 Implementation in libFAUDES

The test for decentralized diagnosis of a modular plant G consisting of m subplants G_j, $j \in \mathcal{I}$, is called by

```
bool IsDecentralizedDiagnosable(const vector<cGenerator>& G, const
    vector<cGenerator>& K, const vector<EventSet>& Σ̂, string& rReport).
```

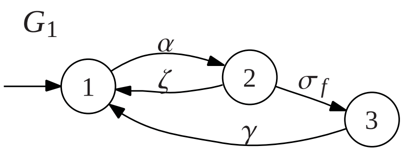
For every subsystem G_j, $j \in \mathcal{I}$, this function

1. creates the a **vector**<**cGenerator**> G_i that contains all the other subsystems, and a **vector**<**EventSet**>& $\hat{\Sigma}_i$ containing all other abstraction alphabets.
2. invokes **bool** IsDecentralizedDiagnosable (**const cGenerator**& G_j, **const cGenerator**& K_j, **const vector**<**cGenerator**>& G_i, **const vector**<**EventSet**>& $\hat{\Sigma}_i$, **string**& rReport). This function
   (a) verifies a loop-preserving observer for every G_i,
   (b) evaluates $\hat{G}_{\mathcal{I}_j}$ which is the parallel composition of the abstractions of all G_i.
   (c) composes $\hat{\Sigma}_{\mathcal{I}_j}$ as the union of all $\hat{\Sigma}_i$
   (d) calls **bool** IsDecentralizedDiagnosable (**const cGenerator**& G_j, **const cGenerator**& K_j, **const cGenerator**& $\hat{G}_{\mathcal{I}_j}$, **const EventSet**& $\hat{\Sigma}_{\mathcal{I}_j}$, **string**& rReport). This function then
      - finds $\hat{\Sigma}_j$ and computes $\hat{G}_j$, $\hat{G}$, $\hat{H}_j$ and H_j as described in Algorithm 5.1,
      - checks if every cycle in $\hat{G}$ contains at least one event in $\hat{\Sigma}_j$,

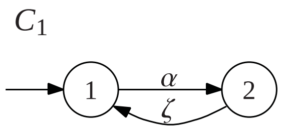
- calls **bool** IsDiagnosable(**const cGenerator**& H_j, **const cGenerator**& K_j, **string**& rReport) to check if H_j is diagnosable with respect to K_j,

and returns the diagnosability property as a boolean value.

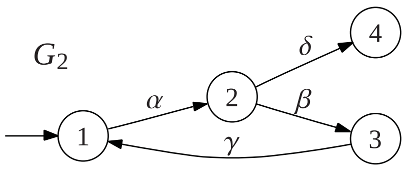
If decentralized diagnosability holds for every subsystem G_j with respect to its local specification K_j, then it also holds for the overall system G with respect to K and *true* is returned; otherwise the result is *false*.



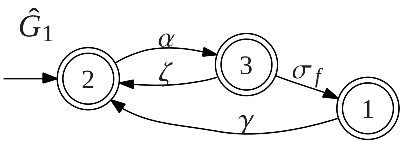
(a) The subsystem G_1.



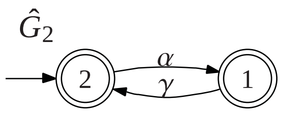
(b) The specification C_1.



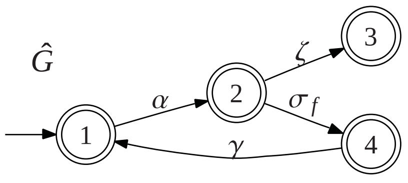
(c) The subsystem G_2.



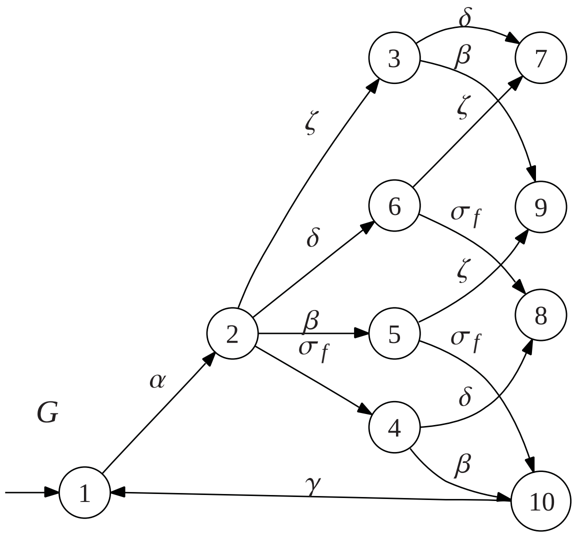
(d) $\hat{G}_1$ is an L-observer of G_1.



(e) $\hat{G}_2$ is not an L-observer of G_2.

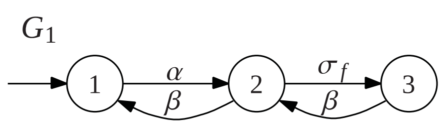


(f) $\hat{G} = \hat{H}_1 = H_1$.

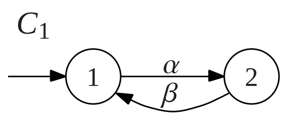


(g) The overall system $G = G_1 \parallel G_2$.

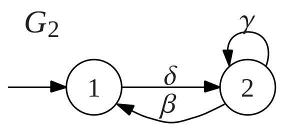
Figure 5.2: Example illustrating necessity of L-observer condition. G_1 and G_2 are locally diagnosable, but $\hat{G}_2$ is not an L-observer of G_2. Thus, the δ-transition deadlocks in the overall system G and this violates diagnosability.



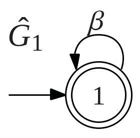
(a) The subsystem G_1.



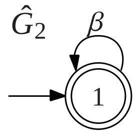
(b) The specification automaton C_1.



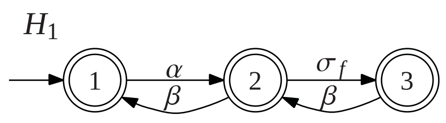
(c) The subsystem G_2.



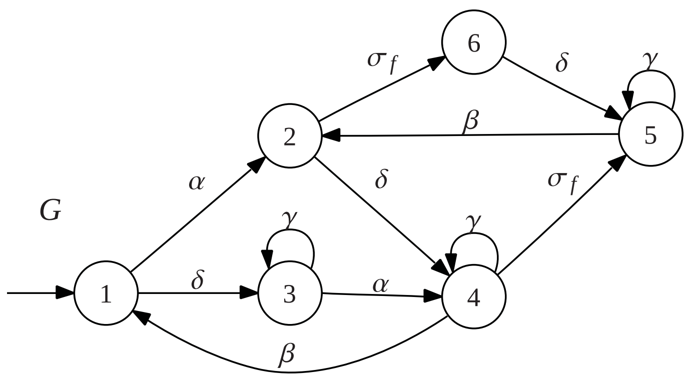
(d) $\hat{G}_1$.



(e) $\hat{G}_2$.

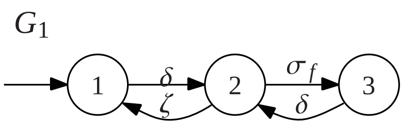


(f) H_1.

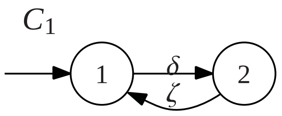


(g) $G = G_1 \parallel G_2$.

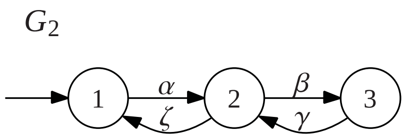
Figure 5.3: Example illustrating necessity of abstractions that are loop-preserving observers. p_2 is not a loop-preserving observer and thus, $\hat{G}_2$ does not contain the γ-loop that violates diagnosability in the overall system $G_1 \parallel G_2$.



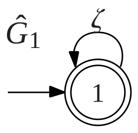
(a) The subsystem G_1.



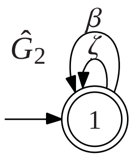
(b) The specification automaton C_1.



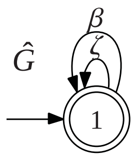
(c) The subsystem G_2.



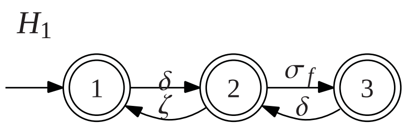
(d) $\hat{G}_1$.



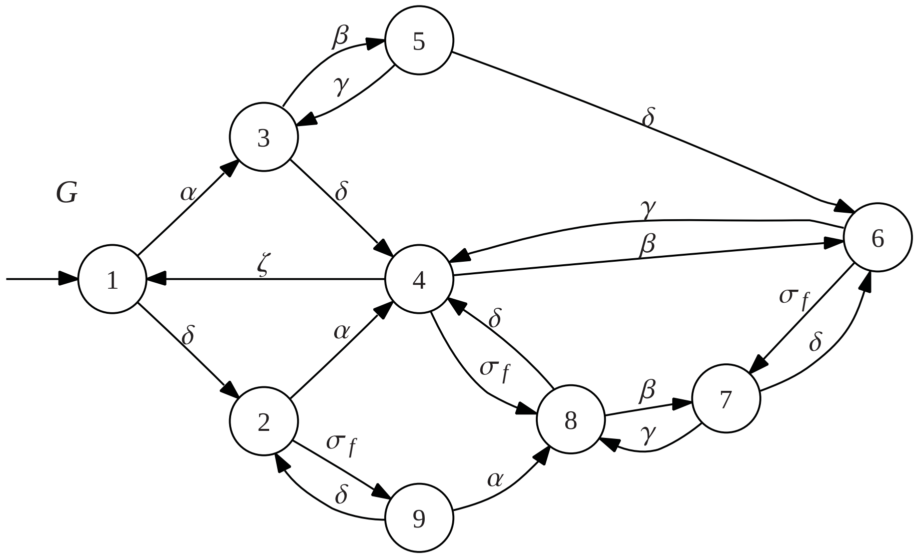
(e) $\hat{G}_2$.



(f) $\hat{G}$.



(g) H_1.



(h) The overall system G.

Figure 5.4: Example illustrating the necessity of the condition that loops in $\hat{G}$ should contain an Event from $\hat{\Sigma}_1$. G_2 is assumed to be decentralized diagnosable and H_1 is generalized diagnosable with respect to K_1, but there exists a β self-loop in $\hat{G}$ and $\beta \notin \hat{\Sigma}_1$. From (h) one can see that the cycle of β and γ at states 7 and 8 violates diagnosability.

Chapter 6

Application of Decentralized Failure Diagnosis to a Fischertechnik Model of an Automated Manufacturing System

In this chapter we apply the approach of testing decentralized diagnosability that has been developed in Chapter 5 to two subsystems of a Fischertechnik simulation model which is available at the *Chair of Automatic Control*. It represents a distributed manufacturing system consisting of a stack feeder, conveyor belts, pushers, rotary tables, production cells and a rail transport system. The Fischertechnik model processes workpieces that are symbolized by wooden blocks. The manufacturing process starts from the stack feeder which delivers the blocks to the first conveyor belt. Figure 6.1 shows a bird's eye view of the Fischertechnik model and Figure 6.2 focuses of the stack feeder and the first part of the conveyor belt. The DES models of these two subsystems are taken from [13] and will be used to perform a test for decentralized diagnosability.

The stack feeder (sf) consists of a stack holding the workpieces and a belt with a tiny block that shoves one workpiece at a time onto the conveyor belt (cb1). The events `sfmv` (stack feeder move) and `sfstp` (stack feeder stop) trigger the motion of the stack feeder belt. A photoelectric barrier detects the presence of a workpiece: At the arrival of a workpiece, `sfwpar` (workpiece arrives) occurs and when it leaves the sensor triggers `sfwplv` (workpiece leaves). The block has got a defined rest position which is detected by a magnetic sensor that triggers `sfr` (rest position) and `stnr` (not in rest position). A possible failure that can occur within the stack feeder is that a workpiece jams. Suppose that after `sfmv`, due to some irregularity, the workpiece lifts a bit at its front site. Then it can hit the lateral cross beam of the stack feeder and block the movement of the stack feeder belt. We call this failure `f_wpjm` (workpiece jams). The *controlled* behaviour of the stack feeder including possible occurrences of the failure `f_wpjm` is shown as automaton G_{sf} in Figure 6.3. The event `sf-cb1` is a shared event between the stack feeder sf and the conveyor belt

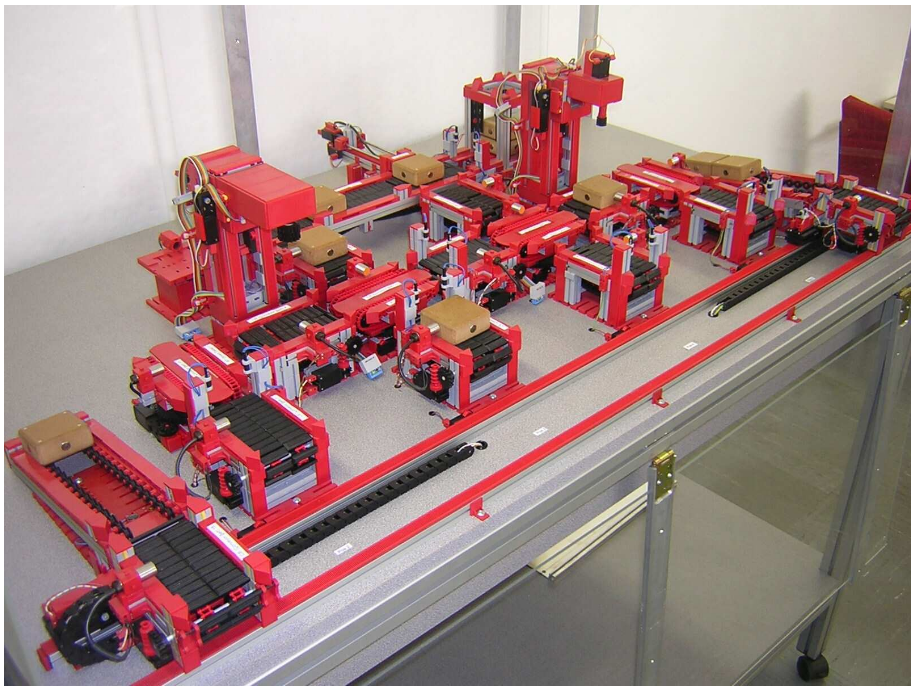


Figure 6.1: The Fischertechnik model of a manufacturing system.

cb1. Whenever it can be executed it indicates that interaction of the stack feeder with the conveyor belt is possible, i. e., a workpiece can be delivered to cb1. To recognize the failure after the jam of the workpiece, a timer with an appropriate threshold is introduced. It starts running and resets, respectively after each occurrence of `sfnr`. If the timer overflows before `sfwplv` occurs the event `t_sf` is issued.

The conveyor belt cb1 transports workpieces by moving in the negative x-direction. The movement is triggered by `cb1-x` and stopped by `cb1stp`. The arrival of the workpiece on the conveyor belt is detected by a capacitive sensor in the pusher and is indicated by `cb1awpar` (workpiece arrives). For this example we simplified the behaviour of cb1. The simplified model G_{cb} of the controlled behaviour of the conveyor belt is shown in Figure 6.4. It includes a failure `f_wpfd` which simply consists of the workpiece dropping from the conveyor belt (workpiece falls down). Additionally, we introduce a second timer that resets every time `cb1-x` occurs. If the overflow occurs before `cb1awpar`, than the event `f_wpfd` is triggered.

G_{sf} and G_{cb}, with their alphabets Σ_{sf} and Σ_{cb} are now considered as two modules of an overall system G. The nominal behaviour of the two subsystems is given by the specifications K_{sf} and K_{cb} that are modelled as specification automata C_{sf} and C_{cb} (see Figure 6.5 and 6.6), where $K_{\text{cb}} = L(C_{\text{cb}})$ and $K_{\text{sf}} = L(C_{\text{sf}})$.

We now want to verify decentralized diagnosability of the overall system $G = G_{\text{sf}} \parallel G_{\text{cb}}$ with respect to the specification $K = K_{\text{sf}} \parallel K_{\text{cb}}$. Therefore, we have to define abstraction alpha-

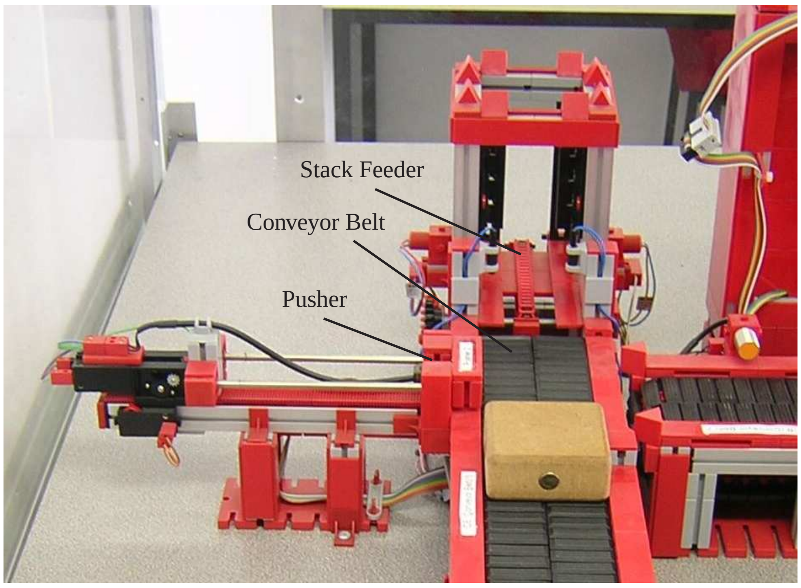


Figure 6.2: Stack feeder and conveyor belt with pusher of the Fischertechnik model.

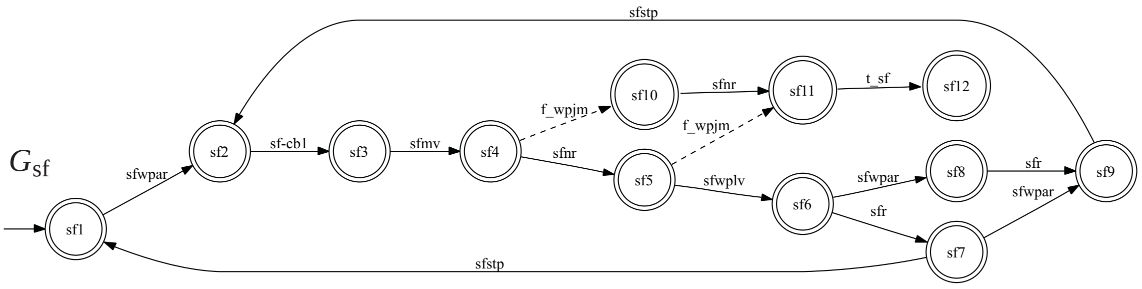


Figure 6.3: The stack feeder model G_{sf} including the unobservable failure event `f_wpjm`.

bets $\hat{\Sigma}_{\text{sf}}$ and $\hat{\Sigma}_{\text{cb}}$. Given the set of shared events $\Sigma_\cap = \Sigma_{\text{sf}} \cap \Sigma_{\text{cb}} = \texttt{sf-cb1}$ and $i = \{\text{sf}, \text{cb}\}$, we choose $\hat{\Sigma}_i$ such that $\hat{\Sigma}_i \supseteq \Sigma_\cap$ are as small as possible but such that the natural projections $p_i\colon \Sigma_i^* \to \hat{\Sigma}_i^*$ are loop-preserving observers for the local plants G_i. In this example it results that $\hat{\Sigma}_{\text{sf}} = \{\texttt{sf-cb1}, \texttt{sfwplv}, \texttt{f\_wpjm}\}$ and $\hat{\Sigma}_{\text{cb}} = \{\texttt{sf-cb1}, \texttt{cb1awpar}, \texttt{f\_wpfd}\}$.

To perform the actual test of diagnosability, the subsystem and specification automata, as well as the abstraction alphabets are stored in *STL vectors* and the function

```
bool IsDecentralizedDiagnosable(const vector<cGenerator>& G, const
    vector<cGenerator>& C, const vector<EventSet>& Σ̂, string& rReport)
```

is called with an additional *string* variable that provides human readable information in case of a negative test result.

As explained in Chapter 5.3, a decentralized diagnosability test is made for every subsystem.

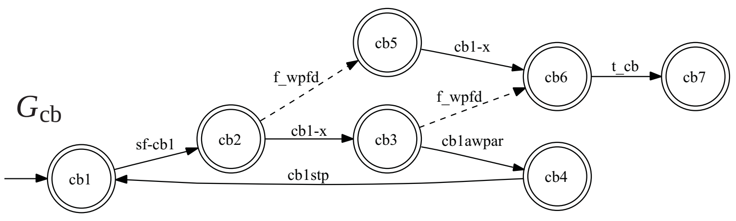


Figure 6.4: Conveyor belt model G_{cb} including the unobservable failure event `f_wpfd`.

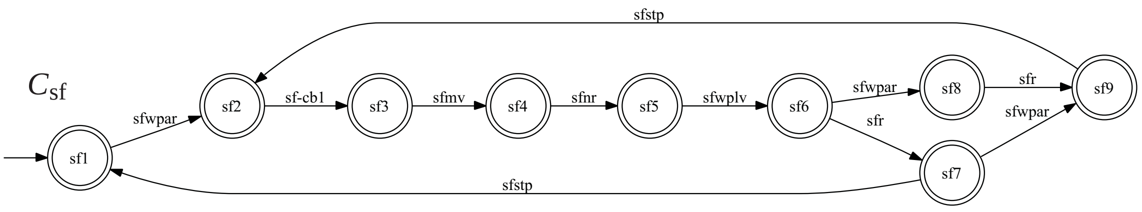


Figure 6.5: The specification automaton C_{sf} for the stack feeder.

First, decentralized diagnosability of G_{sf} is examined by **bool** IsDecentralizedDiagnosable (**const cGenerator**& G_{sf}, **const cGenerator**& K_{sf}, **const cGenerator**& G_{cb}, **const EventSet**& $\hat{\Sigma}_{\text{cb}}$, **string**& rReport) according to the following steps:

1. The abstraction alphabet $\hat{\Sigma}_{\text{sf}}$ is computed such that $p_{\text{sf}} : \Sigma_{\text{sf}}^* \to \hat{\Sigma}_{\text{sf}}^*$ is an L-observer. Here, $\hat{\Sigma}_{\text{sf}} = \{\texttt{sf-cb1}, \texttt{sfwplv}, \texttt{f\_wpjm}\}$. Figure 6.7 shows the abstraction automaton $\hat{G}_{\text{sf}}$.
2. The abstraction $\hat{G}_{\text{cb}}$ is created by performing the loop-preserving projection of G_{cb} on the abstraction alphabet $\hat{\Sigma}_{\text{cb}}$ (see Figure 6.8).
3. $\hat{G}_1 = \hat{G}_{\text{sf}} \parallel \hat{G}_{\text{cb}}$ is the joint behaviour of the abstractions of the subsystems and is depicted in Figure 6.9.
4. $\hat{H}_{\text{sf}}$ is the local view of $\hat{G}_1$ from the local site G_{sf}. It is created by replacing all transitions in $\hat{G}_1$ with events that are not in $\hat{\Sigma}_{\text{sf}}$ by ε-transitions (see Figure 6.10). Note that ε-transitions do not exist in libFAUDES. We use **void** cProjectNonDet (**cGenerator**& $\hat{G}_1$,**const EventSet**& $\hat{\Sigma}_{\text{sf}}$) that replaces ε-transitions by inserting additional transitions and further initial states if necessary.
5. $H_{\text{sf}} = G_{\text{sf}} \parallel \hat{H}_{\text{sf}}$ is computed. This parallel composition incorporates the high-level behaviour of G_{cb} in H_{sf} (see Figure 6.11).

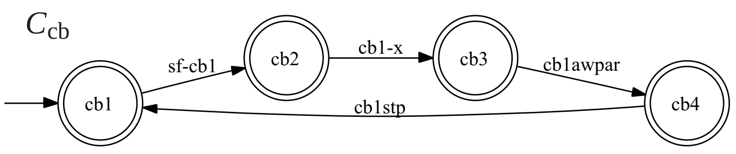


Figure 6.6: The specification automaton C_{cb} for the conveyor belt.

6. The function determines that H_{sf} is generalized diagnosable with respect to K_{sf} and additionally verifies that every cycle in $\hat{G}_1$ contains at least one event in $\hat{\Sigma}_{\text{sf}}$. From Proposition 5.1 it follows, that G_{sf} is decentralized diagnosable with respect to K_{sf}.

To illustrate the testing according to Proposition 5.1, examine H_{sf} and G_1: Firstly, from the model of H_{sf}, generalized diagnosability with respect to K_{sf} can be recognised. All faulty traces containing the failure event `f_wpjm` (cp. the faulty states within {8,9,10,19,20,21,28,29,30}) do deadlock with the occurrence of `t_sf` in any of the states {10,21,30}. Since `t_sf` only occurs in the faulty traces, the violation of K_{sf} can be determined unambiguously. Secondly, looking at $\hat{G}_1$ we identify two cycles at states {1,2,3} and {1,2,5}, respectively. Both of them contain the events `sf-cb1` and `sfwplv` which belong to $\hat{\Sigma}_{\text{sf}}$.

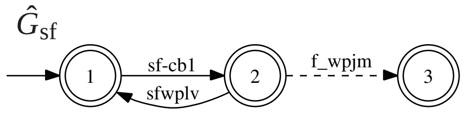


Figure 6.7: The abstraction automaton $\hat{G}_{\text{sf}}$.

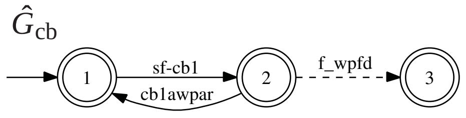


Figure 6.8: The loop-preserving abstraction automaton $\hat{G}_{\text{cb}}$.

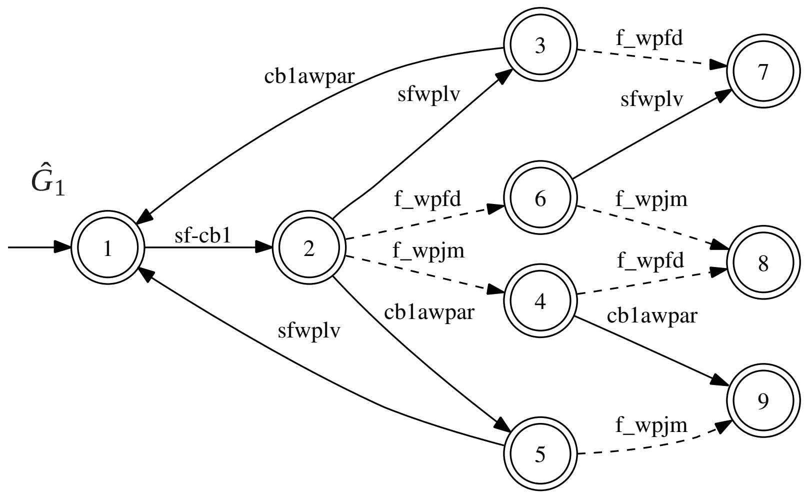


Figure 6.9: $\hat{G}_1$ represents the joint behaviour of the abstractions $\hat{G}_{\text{sf}}$ and $\hat{G}_{\text{cb}}$.

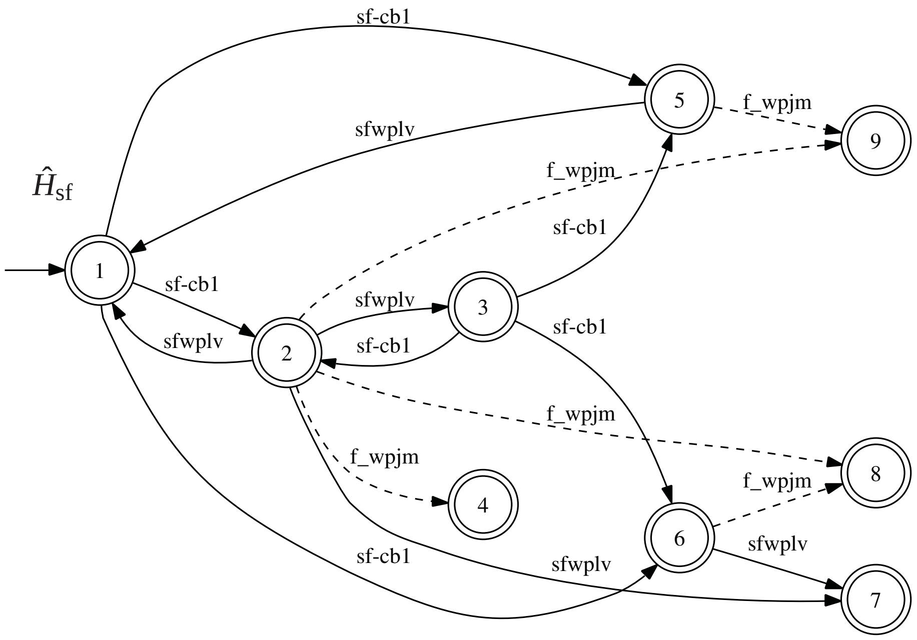


Figure 6.10: $\hat{H}_{\mathrm{sf}}$ is the local view of $\hat{G}_1$ from G_{sf}.

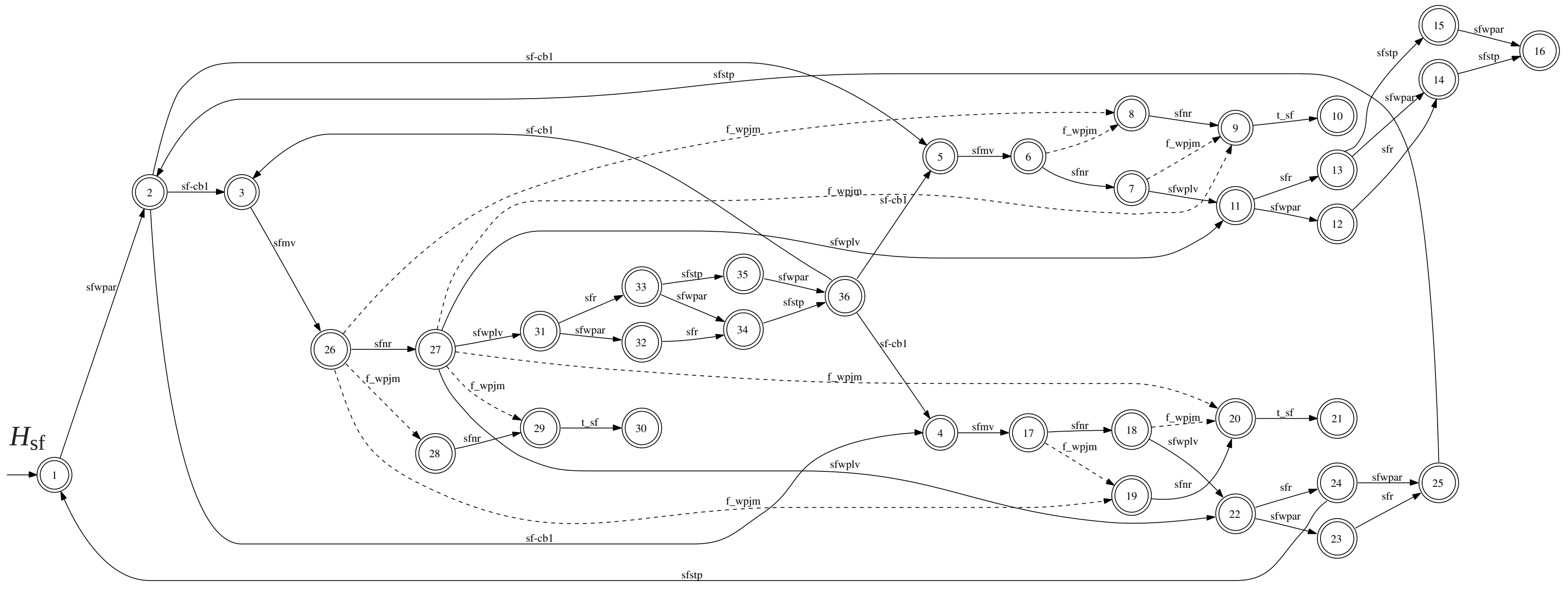


Figure 6.11: H_{sf} is the behaviour of G_{sf} incorporationg the high-level behaviour of G_{cb}.

Next, decentralized diagnosability of the second subsystem, namely G_{cb} is checked analogously. The testing automata are shown in Figures 6.12 to 6.15. Again, H_{cb} turns out to be generalized diagnosable with respect to K_{cb}, as similarly to before, every faulty trace unambiguously ends with the occurrence of `t_cb` that leads into any of the deadlocking states in {8,14,19}. Additionally, any of the cycles occurring at states {1,2,3} and {1,2,5} in $\hat{G}_2$ contains at least one event in $\hat{\Sigma}_{\mathrm{cb}}$. It follows from Proposition 5.1 that G_{cb} is decentralized diagnosable with respect to K_{cb}.

All in all, both subsystems G_{sf} and G_{cb} are individually decentralized diagnosable. Thus, it follows from Theorem 5.1 that the overall system $G = G_{\mathrm{sf}} \parallel G_{\mathrm{cb}}$ is decentralized diagnosable with respect to the specification $K = K_{\mathrm{sf}} \parallel K_{\mathrm{cb}}$.

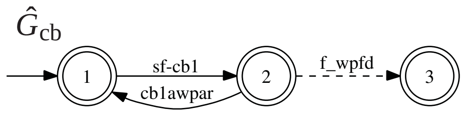


Figure 6.12: The abstraction automaton $\hat{G}_{\mathrm{cb}}$.

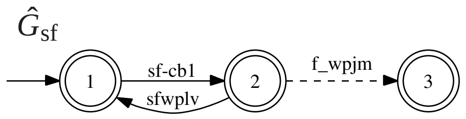


Figure 6.13: The loop-preserving abstraction automaton $\hat{G}_{\mathrm{sf}}$.

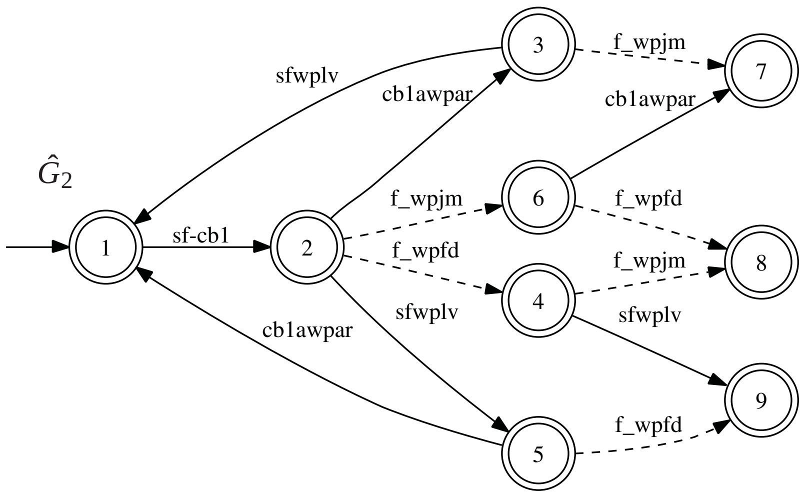


Figure 6.14: $\hat{G}_2$ represents the joint behaviour of the abstractions $\hat{G}_{\mathrm{sf}}$ and $\hat{G}_{\mathrm{cb}}$.

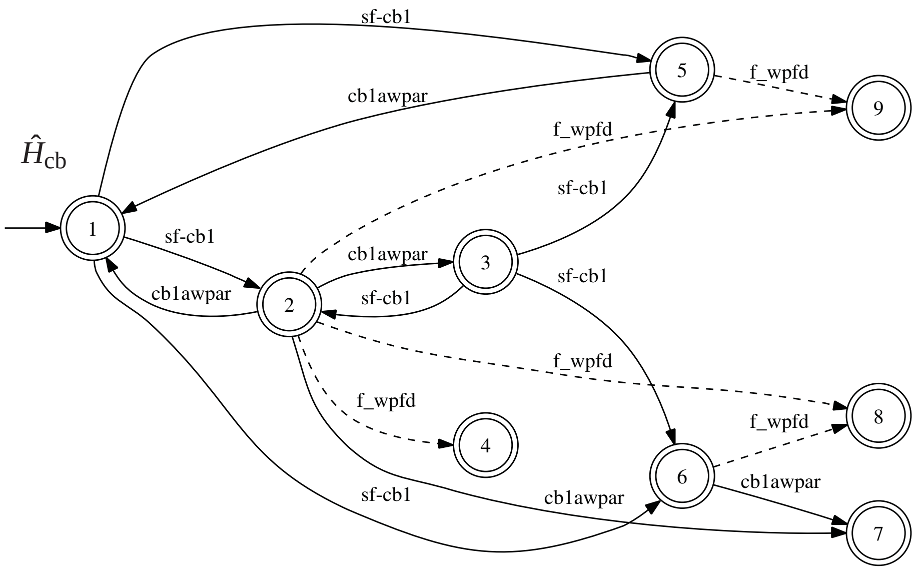


Figure 6.15: $\hat{H}_{\text{cb}}$ is the local view of $\hat{G}_2$ from G_{cb}.

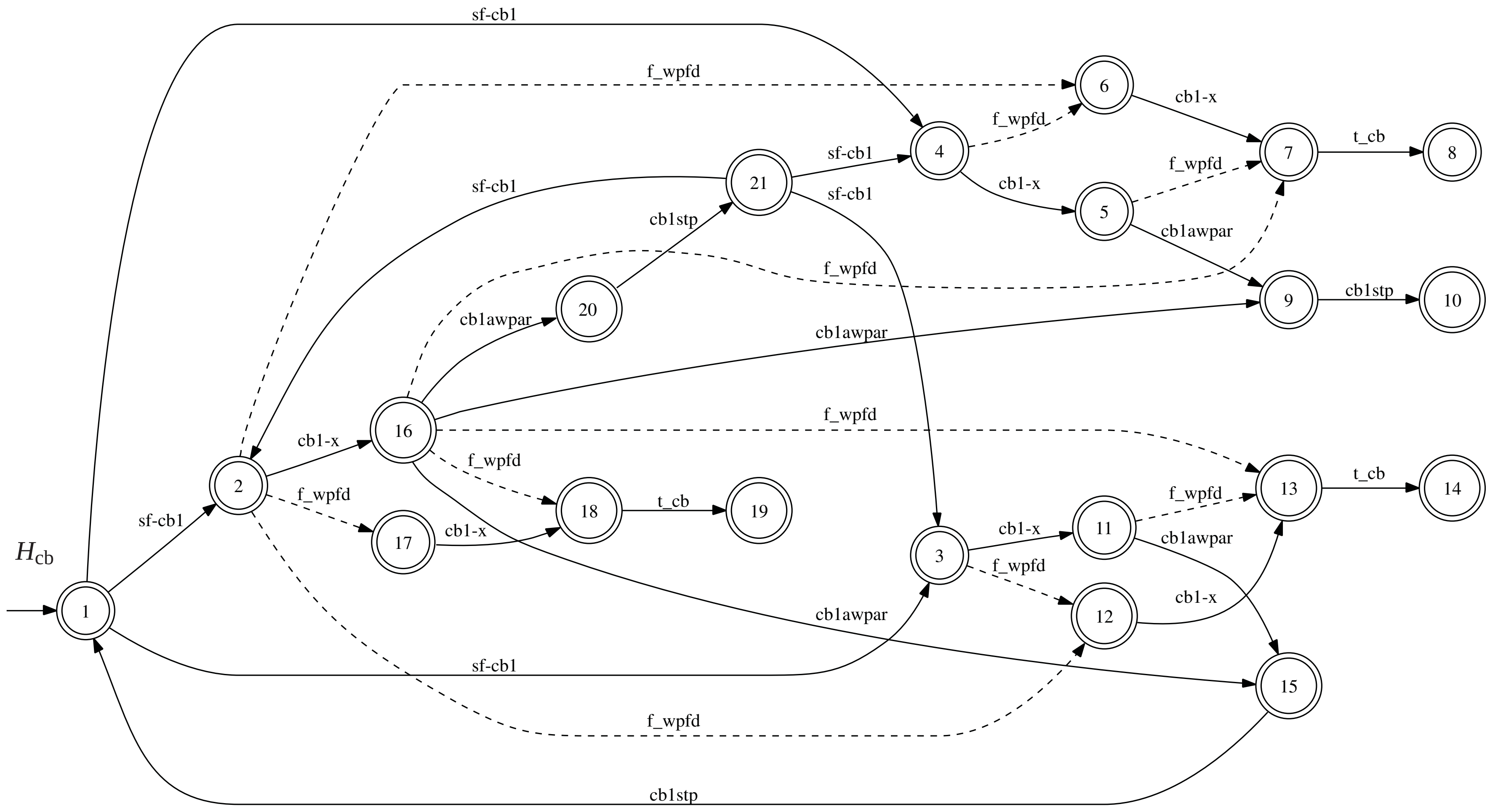


Figure 6.16: H_{cb} is the behaviour of G_{cb} incorporationg the high-level behaviour of G_{sf}.

sf	states	**cb**	states	**sf**$\|\|$**cb**	states
G_{sf}	12	G_{cb}	7	G	84
C_{sf}	9	C_{cb}	4	C	36
H_{sf}	36	H_{cb}	21	–	–
$\tilde{G}_{\text{sf}}$	36	$\tilde{G}_{\text{cb}}$	21	$\tilde{G}$	84

Table 6.1: The table lists the relevant automata and the cardinality of their state sets to compare the test for decentralized diagnosability with the test for language diagnosability.

To end with, we compare the test of decentralized diagnosability of G_{sf} and G_{cb} with the test of diagnosability for the overall system $G = G_{\text{sf}} \parallel G_{\text{cb}}$ with respect to $K = L(C)$ with $C = C_{\text{sf}} \parallel C_{\text{cb}}$.

For the test of decentralized diagnosability, the automata $\hat{G}_i$, $\hat{G}$, $\hat{H}_i$, and H_i have to be generated for every subsystem and then H_i is tested for diagnosability with respect to K_i.

In contrast, for the centralized test, G and C are computed as parallel compositions and then G is tested for diagnosability against K. For an impression of the system G and the specification automaton C, Figure 6.17 shows $\tilde{G}$ which has got the same transition structure as G (but marks its faulty language) and Figure 6.18 depicts C. Table 6.1 compares the sizes of the automata by listing the state numbers of the automata in the decentralized diagnosis test compared to those of the centralized diagnosability test.

Summarized, the difference in size of the automata in the decentralized test to those of the centralized test is evident: While the largest automaton for the centralized test has 84 states, the largest automaton for the decentralized test has only 36 states. Since complexity reduction is already observed in this example, we expect it to be even more visible in the case of large-scale systems with various subsystems.

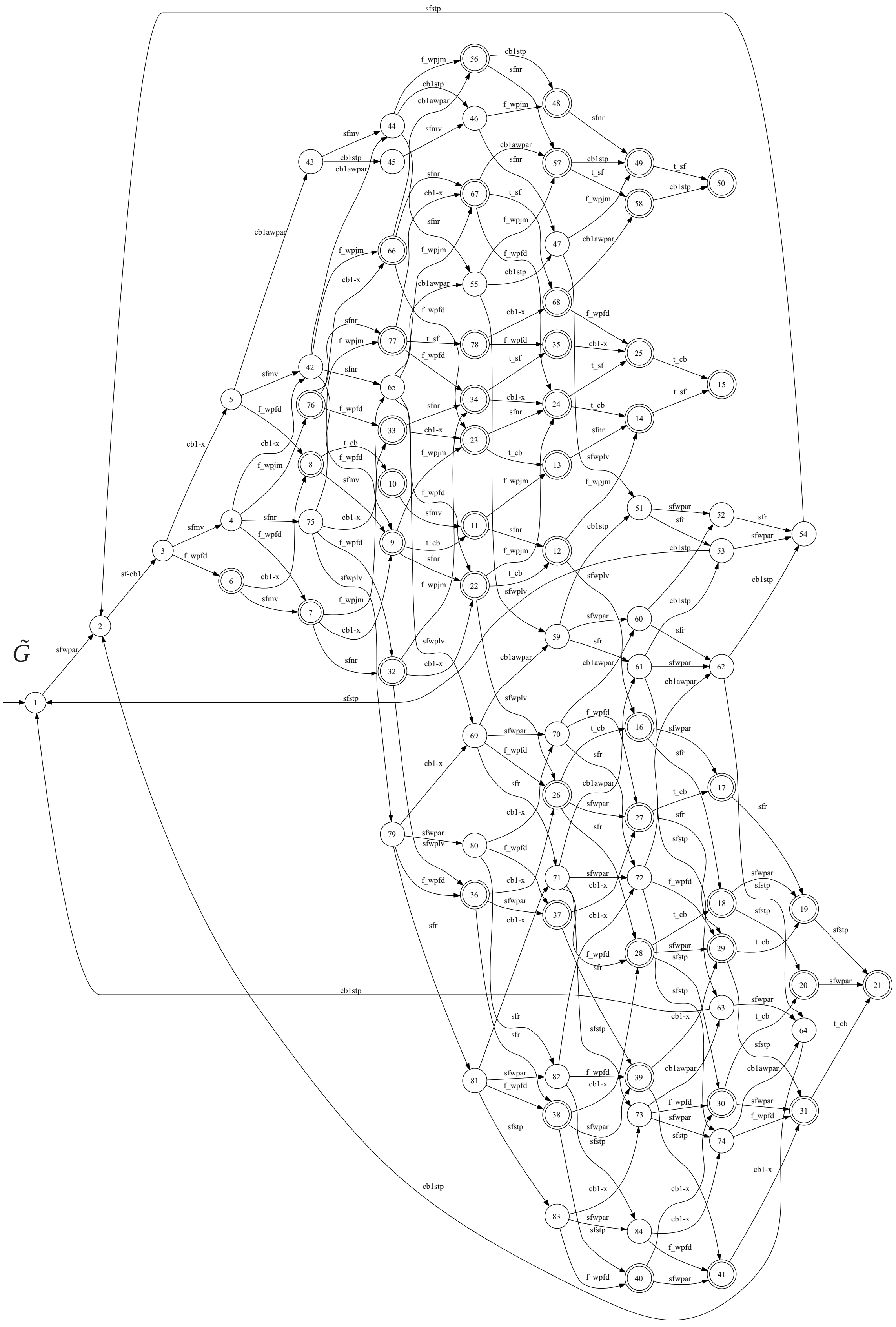


Figure 6.17: $\tilde{G}$ marks the faulty language of the overall system $G = G_{\text{sf}} \parallel G_{\text{cb}}$.

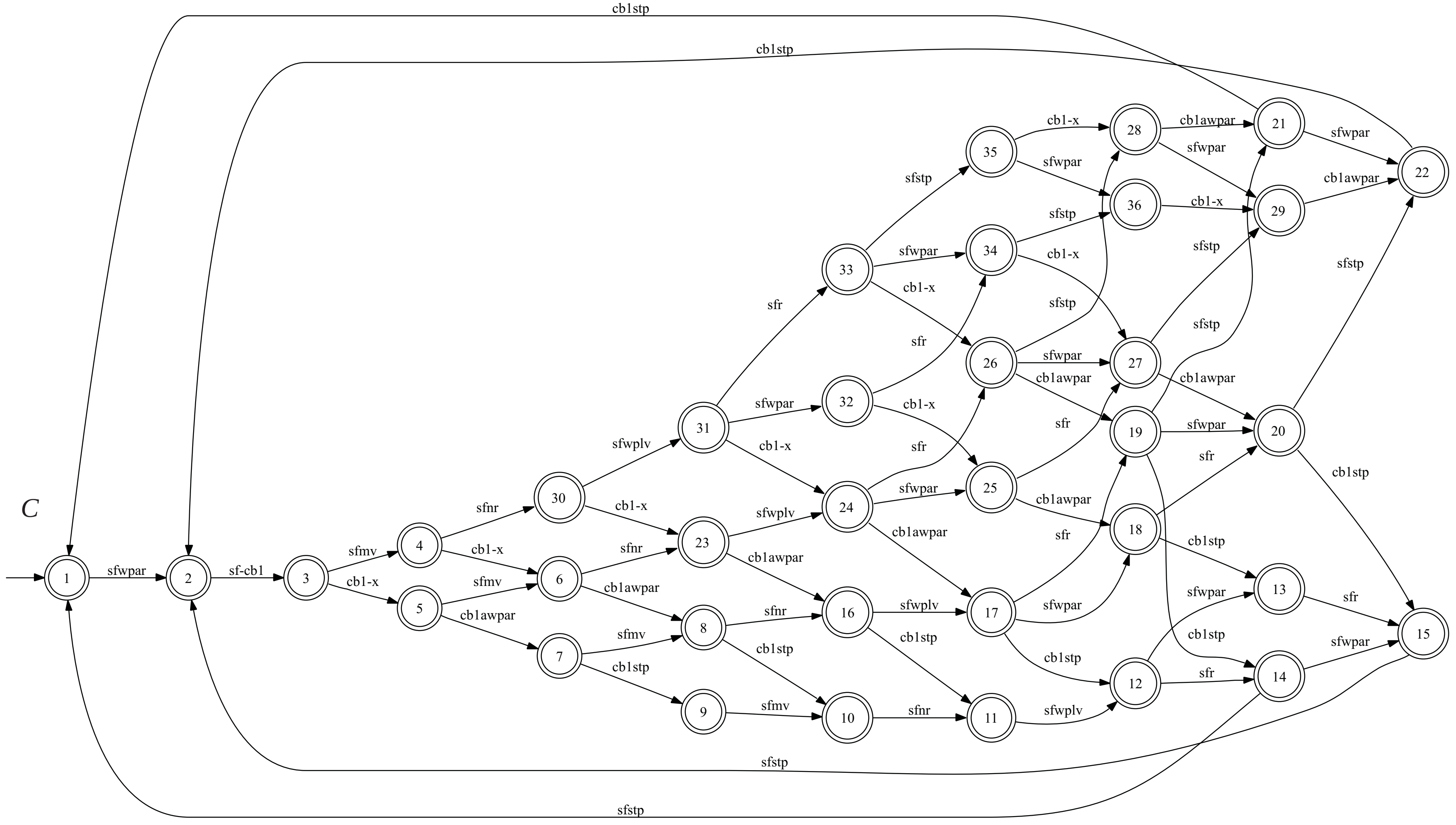


Figure 6.18: $C = C_{\text{sf}} \parallel C_{\text{cb}}$ is the overall specification automaton.

Chapter 7

Conclusion

Failure diagnosis has become a crucial task within the research of discrete events systems (DES) since the middle of the 1990s. With systems growing in size and complexity it becomes important to find efficient procedures and algorithms for the on-line diagnosis and the validation of diagnosability itself. In this thesis we focused on the latter.

After introducing the relevant basics of DES, we presented the notions of diagnosability and I-diagnosability, and the diagnoser according to Sampath *et al.* [12]. A method for testing diagnosability with polynomial complexity was given according to Jiang *et al.* [5] and we additionally developed a polynomial-time test for verifying the I-diagnosability property of a system.

In accordance with the notion of codiagnosability of Qiu and Kumar [8] we introduced diagnosability with respect to specification languages and developed a test for the validation of this so-called language diagnosability which is polynomial in complexity.

In Chapter 5, we presented a polynomial-time abstraction-based approach for decentralized diagnosis of a modular system with local specifications which does not require the construction of the complete plant model and specification. To determine decentralized diagnosability for a subsystem, the local view of the abstractions joint behaviour is determined as a nondeterministic automaton. From there a model of the subsystem that incorporates an abstraction of the other subsystems' behaviour is computed. This newly obtained model is then used to perform a generalized diagnosability test with respect to the local specification. The notion of decentralized diagnosability for an individual subsystem was extended to the overall system and the practicability of the required conditions was illustrated.

The structure of the diagnoser, the diagnosability tests with respect to failure events, the diagnosability test with respect to a specification, and the procedure of decentralized diagnosability testing have been implemented as a plug-in for the C++ software library libFAUDES.

Finally, we validated the functionality of the newly introduced approach and the algorithmic implementation by performing a decentralized diagnosis test for two interacting subsystems of a Fischertechnik model of a manufacturing system.

In future work, further tasks in the scope of failure diagnosis could be the implementation of online diagnostics in the libFAUDES library, creating an algorithm that extends high-level alphabets such that projections on these alphabet become loop-preserving or adapting the failure models of a system such that it becomes decentralized diagnosable. Moreover, decentralized diagnosability testing could be studied for modular systems with a global specification that is defined as part of the high-level language of the system.

Bibliography

[1] libFAUDES (Friedrich-Alexander University Discrete Event Systems Library). http://www.rt.eei.uni-erlangen.de/FGdes/faudes/, 2009.

[2] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison Wesley, January 1975.

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

[4] Lei Feng and W.M. Wonham. On the computation of natural observers in discrete-event systems. *Decision and Control, 2006 45th IEEE Conference on*, pages 428–433, Dec. 2006.

[5] Shengbing Jiang, Zhongdong Huang, V. Chandra, and R. Kumar. A polynomial algorithm for testing diagnosability of discrete-event systems. *Automatic Control, IEEE Transactions on*, 46(8):1318–1321, Aug 2001.

[6] Andrea Paoli and Stéphane Lafortune. Diagnosability analysis of a class of hierarchical state machines. *Discrete Event Dynamic Systems*, 18(3):385–413, 2008.

[7] W. Qiu, Q. Wen, and R. Kumar. Decentralized diagnosis of event-driven systems for safely reacting to failures. *Automation Science and Engineering, IEEE Transactions on*, 6(2):362–366, April 2009.

[8] Wenbin Qiu and R. Kumar. Decentralized failure diagnosis of discrete event systems. *Systems, Man and Cybernetics, Part A: Systems and Humans, IEEE Transactions on*, 36(2):384–395, March 2006.

[9] P.J. Ramadge and W.M. Wonham. Supervisory control of a class of discrete event systems. *SIAM J. Control and Optimization*, 25:206–230, 1987.

[10] P.J. Ramadge and W.M. Wonham. The control of discrete event systems. *Proceedings of the IEEE*, 77:81–98, 1989.

[11] M. Sampath, R. Sengupta, S. Lafortune, K. Sinnamohideen, and D. Teneketzis. Failure diagnosis using discrete event models. *Decision and Control, 1994., Proceedings of the 33rd IEEE Conference on*, 3:3110–3116 vol.3, Dec 1994.

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

[13] K. Schmidt. *Hierarchical Control of Decentralized Discrete Event Systems: Theory and Application*. PhD thesis, Lehrstuhl für Regelungstechnik, Universität Erlangen-Nürnberg, 2005. Download: http://www.rt.eei.uni-erlangen.de/FGdes/publications.html.

[14] R. Su and W. Wonham. Hierarchical fault diagnosis for discrete-event systems under global consistency. *Discrete Event Dynamic Systems*, 16(1):39–70, January 2006.

[15] K. C. Wong and W. M. Wonham. On the computation of observers in discrete-event systems. *Discrete Event Dynamic Systems*, 14(1):55–107, 2004.

[16] Tae-Sic Yoo and S. Lafortune. Polynomial-time verification of diagnosability of partially observed discrete-event systems. *Automatic Control, IEEE Transactions on*, 47(9):1491–1495, Sep 2002.

[17] C. Zhou, R. Kumar, and R.S. Sreenivas. Decentralized modular diagnosis of concurrent discrete event systems. *Discrete Event Systems, 2008. WODES 2008. 9th International Workshop on*, pages 388–393, May 2008.