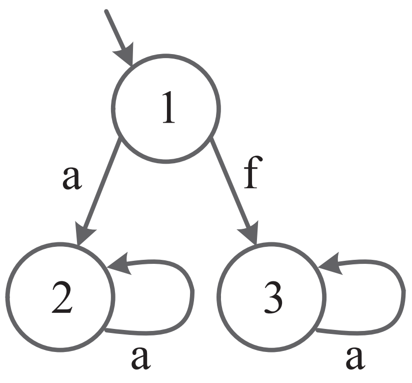
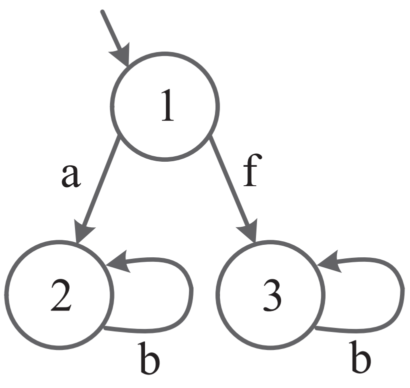
CHALMERS



(a) Non-diagnosable automaton



(b) Diagnosable automaton

Diagnosis of Discrete Event Systems

Mona Noori Hosseini

Supervisor: Bengt Lennartson

Department of Signal and System
Automation Group
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2011

Master's Thesis EX100/2011

Diagnosis of Discrete Event Systems

Master's Thesis EX100/2011

Department of Signal and System
Chalmers University of Technology
SE-41296 Göteborg
Sweden

Tel. +46-(0)31 772 1000

Prepared With LaTeX
Reproservice / Department of Signal and System
Göteborg, Sweden 2011

Abstract

In many applications where a Discrete Event System (DES) contains unobservable events, determining whether certain unobservable or failure events have occurred is of interest. This is the problem of failure diagnosis, which has attracted much attention to this field. Earlier work on diagnosis have mainly based on ordinary finite automata.

In this thesis, failure diagnosis of discrete event systems based on nondeterministic Extended Finite Automaton (EFA) is studied to gain more compactness and ease of implementation. For this purpose, the diagnosability test on Extended Finite Automata (NEFAs) is performed. Moreover, guards and actions are replaced by conditions, involving both the current and the next value of variables, which are augmented to the transitions.

In this perspective, two computation tree logic (CTL) formulas are proposed as proper specifications for the diagnosability verification. NuSMV, as a symbolic model checker software tool, is then utilized to check the specification, in order to verify the diagnosability of the synchronized NEFAs.

Keywords: Diagnosability, Failure diagnosis, Discrete event systems, Nondeterminism, Extended finite automata (EFAs)

Acknowledgements

First of all, I would like to acknowledge all the support from my supervisor, Prof. Bengt Lennartson, who has always been the inspiration to carry on the research with his encouragements and never left me alone either in the very hectic days of the research or in the days that I was completely stuck facing many different problems. Thank you for not letting me get discouraged by always emphasizing the fact that we could handle the problems and reducing the stress during the whole project period. Also, thank you for all your directions that really made the project easy to deal with. It is pretty fruitful and fun working with you.

Special thanks to Alexey Voronov for his kind guidance to solve software problems. I wish to extend my warmest thanks to all professors and students that have helped me in automation group at Chalmers University of Technology.

I want to thank my parents who always encourage me in life with all their support and love from the bottom of their hearts. Last but definitely not least, I would also like to thank Behrooz for his love and support. Sitting beside you while doing my thesis was one of the best experiences I had.

Mona Noori Hosseini, Göteborg 2011

Contents

1

Introduction

Many industrial systems are categorized as discrete event systems, where their behavior is monitored by observing some system events, recordable by sensors. These events are called observable events. On the other hand, there are also non-recordable events in a system, normally denoted unobservable. The existence of unobservable events adds ambiguity to the system, since executing unobservable events means that the system cannot recognize in which state it is. Moreover, among the unobservable events, some failures may happen, which are deviations of the system from its normal or required behavior. Beside failure events, what is called failure can also be visiting a wrong state or not fulfilling a desired specification.

In many applications where a system contains unobservable events, determining whether certain unobservable or failure events have occurred is of interest. This is the problem of failure diagnosis, which has attracted much attentions to this field. Failure diagnosis is to determine the occurrence of a failure that has happened in the past, through the observation of a bounded trace of observable events.

In this perspective, the notion of diagnosability is the ability to detect every failure occurrence in a system. This notion is introduced in [1, 2], where necessary and sufficient conditions for diagnosability and I-diagnosability are provided. The notion of I-diagnosability indicates that a system is I-diagnosable if it is possible to diagnose failures, not always, but whenever the failure events are followed by certain observable indicator events that are associated with the failures. There, a diagnoser needs to be completely constructed in order to test the diagnosability of model. The proposed diagnosability algorithm has exponential complexity in the number of states. In [1, 2], the concepts of uncertain and indeterminate cycles are described as the basis for the diagnosability analysis of automata.

In [3], an integrated approach for control and diagnosis is presented, which is called active diagnosis. Active diagnosis is an approach for design of diagnosable systems by appropriate design of a system controller. On the other hand, the term passive diagnosis implies the role of diagnoser as a simple observer of system behavior for potential failures. In [4], a polynomial algorithm is proposed where, in contrast to [1], there is no need to construct a diagnoser beforehand. Moreover, in [5] the authors have extended the diagnosability test in [4] to rule-based models where an online diagnosis algorithm is also introduced.

To motivate the importance of diagnosability analysis it is fruitful to explain it through a real industrial example. Consider a motor vehicle whose wheels are equipped with an Anti-lock Braking System (ABS). The main purpose in using ABS is to prevent the wheels of the vehicle from locking up while braking. In modern cars, each wheel has a local ABS which works independently from other wheels. One of the failures which may occur in the system is the stuck-at-on failure. It means that a sensor, that is responsible for activation of ABS, permanently observes a locking condition on the wheel regardless of its actual condition. This causes ABS to remain active even in the case that the wheel is not locked. The other failure type which may be seen in an ABS is stuck-at-off failure, which means that a sensor permanently observes a non-locking condition on the wheel regardless of its actual condition. In this case ABS never become activated even in the case that the wheels are locked. It is very crucial to be able to diagnose these two failure types, in order to make a driving safe and to protect cars from slippering and accidents.

With this background, in [6], authors have investigated this issue and focused on a brake-by-wire system combined with a high level brake function, ABS. They worked on diagnosability analysis of a vehicle whose wheels are equipped with an ABS. The behavior of the ABS is modeled by Petri Nets (PNs). Diagnosability analysis has been performed using the method proposed in [7] where a verifier net (VN) is computed and its reachability graph is analyzed. Using these techniques, the failure occurrence in the model can be analyzed locally in a wheel and without considering the wheel in interaction with other wheels. Moreover, it can be analyzed globally in order to check its diagnosability, which in both cases, locally and globally, the stuck-at-on failure is diagnosable. The same algorithm is utilized for cases with stuck-at-off failures which results in that stuck-at-off failures are merely globally diagnosable, and it can not be diagnosed locally in each ABS of wheels. Consequently, this failure diagnosis method enables us to distinguish whether a failure has happened in accordance to the braking system of a motor vehicle.

The above mentioned results are mainly formulated in the ordinary automata or petri net framework. This thesis implements Extended Finite Automaton (EFA) introduced in [8]. Diagnosability test is extended on synchronization of an arbitrary number of EFAs which are augmented with guards and actions on each transition. In this work, instead of guard and action which were used by [8], a condition is attached to each transition which conveys the same concept as in the ordinary EFA.

Furthermore, two computation tree logics (CTLs) are proposed as the proper specifications for diagnosability verification. The two specifications are designed based on the concept of indeterminate cycle which is described in Chapter 4. For this purpose, the specifications are checked in all possible states of the model, in order to see whether the model fulfills or violates the specifications. Thus, a suitable software tool is required for diagnosability verification by testing the specification.

NuSMV is a symbolic model-checker tool which is utilized to verify the specification.

The advantage of this work is that the compactness of EFAs along with the augmented condition relations and the nondeterminism help us to represent large systems in a compact and understandable model. Moreover, it is very crucial to verify the diagnosability of interacting systems and be able to diagnose failure occurrences while a systems is operating.

In this thesis, the notions of the system and specifically Discrete Event Systems (DES) are described in the first chapter. The second chapter illustrates the basic concepts for the rest of the work which are the definition of an ordinary automaton and formal languages. In the third chapter, notion of diagnosability and failure diagnosis is expressed. Moreover, different diagnosability test algorithms and proper temporal logic specifications for model verification are illustrated. The fifth chapter starts by introducing EFAs, nondeterministic EFAs and synchronization of EFAs. In the end, the sixth chapter includes four different NuSMV codes along with their description and algorithms, which are implemented in different parts of the thesis.

2

SYSTEMS

2.1 Introduction

This chapter starts with the explanation of some notions and definitions which are the building blocks of this work.

The Concept of System

System is one of the intuitive concepts that can be understood best by examples rather than an exact definition. In general a system is defined as [9]:

i An aggregation or assemblage of things so combined by nature or man as to form an integral or complex whole (*Encyclopedia Americana*).

ii A regularly interacting or interdependent group of items forming a unified whole (*Webster's Dictionary*).

iii A combination of components that act together to perform a function not possible with any of the individual parts (*IEEE Standard Dictionary of Electrical and Electronic Terms*).

Based on the above definitions, it is deduced that a system consists of a function and some interacting components. A system is influenced by inputs and the resulting behavior can be observed by its outputs. The function and system components along with a set of measurable variables, associated with the system, represent a model for the actual system. Systems can be classified in different categories and groups. For instance, based on the characteristic of the internal behavior, systems are categorized as *static* or *dynamic* systems. Static systems are memoryless and can be represented by algebraic equations, in contrast to dynamic systems. The outputs of dynamic systems are related to the past values of the inputs and the input-output relation is described by differential equations. Dynamic systems are more challenging from an analysis and complexity point of view.

These characteristics are valid for both continuous-time and discrete event systems. Note that continuous-time system inputs and outputs are signals, i.e., temperature, position and voltage. In discrete event systems the inputs and outputs include both signals and discrete events. There are some differences between continuous-time and discrete event systems and their signals and events, which are described in the sequel.

Continuous States

The internal system behavior of a continuous-time dynamic systems is represented by continuous state variables.

Linear time-invariant continuous-time dynamic systems can be described by continuous-time state space models, as follows

$$\begin{aligned} \dot{x}(t) &= f(x(t),u(t),t), \qquad x(t_0) = x_0 \\ y(t) &= g(x(t),u(t),t) \end{aligned} \tag{2.1}$$

where (2.1) includes the set of state and output equations with initial conditions [9].

In continuous-state systems, the state variables can generally take on any real value. The continuous-state models are reduced to the analysis of differential equations.

Discrete States

A system with discrete state space is assumed to change state only at discrete-time instances t_k, k = 1, 2, 3, ... Furthermore, the discrete state x, the input signal u, and the output signal y are restricted to take values from countable discrete sets. An example of such a set is the set of the non-negative integers. The dynamic behavior of a discrete system is often simpler to visualize. The update of the state x at time t_k is determined by the state space model

$$\begin{aligned} x(t_k^+) &= f(x(t_k),u(t_k),t_k) \\ y(t_k) &= g(x(t_k),u(t_k),t_k) \end{aligned} \tag{2.2}$$

which is similar to (2.1), except for the derivative $\dot{x}(t)$ that is replaced by $x(t_k^+) = lim_{\varepsilon \to 0} x(t_k + \varepsilon)$, i.e. the state immediately after the discrete update at time t_k [10].

Discrete-Time Systems

There are several reasons why we might want to use discrete time approach.

1. Any digital computer used as a component in a system operates in discrete-time fashion, that is, it is equipped with an internal discrete-time clock. Whatever variables the computer recognizes or controls are only evaluated at those time instants corresponding to the clock ticks.

2. Many differential equations of interest in the continuous-time models can only be solved numerically by computers. Such computer-generated solutions are actually discrete-time versions of continuous-time functions. Therefore, dealing with discrete-time models is reasonable even if the ultimate solutions are in continuous-time form anyway.

3. Digital control techniques, which are based on discrete-time models, often provide considerable flexibility, speed and cost. This is because of advances in digital hardware and computer technology.

4. Some systems, such as economic models based on data recorded, only at regular discrete intervals, are inherently discrete-time.

2.2 Discrete Event Systems

In cases where the state space can be represented by a discrete set and the state transitions are observable at discrete points of time, these state transitions can be associated with events in discrete event systems.

The Concept of Event

An event causes transitions from one state to another and they occur in a time instant (zero time). In the rest of the text, a general event and a set of events are represented by σ and Σ, respectively.

Time-Driven and Event-Driven Systems

In time-driven systems, the state continuously changes as time changes. In event-driven systems, only the occurrence of asynchronously generated discrete events forces instantaneous state transitions and the state remains unchanged between event occurrences. Modeling and analysis of these systems are more complicated.

2.3 Conclusion

A Discrete Event System (DES) is a system where the state space is a discrete set and the state transition mechanism is event-driven. The state of a DES changes by the occurrence of asynchronous discrete events over time. Many technological systems are discrete event systems.

Remark. Note that discrete event systems are not the same as discrete time systems. Discrete event systems, are modeled in both discrete or continuous time, the same as Continuous-Variable Dynamic Systems (CVDS). That is, discrete time systems contain both the CVDS and the DES. Some examples of DES are queueing, computer, manufacturing and traffic systems.

Finally, the systems on which the thesis concentrates are dynamic, time-invariant, nonlinear (boolean type), discrete state, event driven systems. In the next chapter automata and a formal language are introduced as the main modeling approach for DES along with different operations on such languages. Moreover, nondeterministic automata and synchronous composition of automata are illustrated by examples.

3

Automata and Formal Languages

3.1 Introduction

In the previous chapter, the differences between DESs and CVDSs were described, and it was illustrated that DESs are more appropriate for describing the behavior of higher level systems. There are different modeling formalisms that are available for DESs in comparison to continuous dynamic systems where differential equations is the main modeling formalism. As the main modeling approach for DESs, automata will be introduced in the following section, and a related framework that is formal language will be described in Section 3.3.

3.2 Automata

An intuitive way to describe discrete event systems is to define an automaton model. A deterministic finite state automaton A is a 4-tuple

$$A = \langle Q, \Sigma, T, q_i \rangle \tag{3.1}$$

where

(i) $Q = \{q_1, q_2, \ldots, q_n\}$ is the finite set of states,

(ii) $\Sigma = \{\sigma_1, \sigma_2, \ldots, \sigma_p\}$ is the finite set of events, which is the *alphabet* of the automaton,

(iii) $T : Q \times \Sigma \times Q$ is the transition relation that projects the state to the next one after the execution of the event, and

(iv) q_i is the initial state.

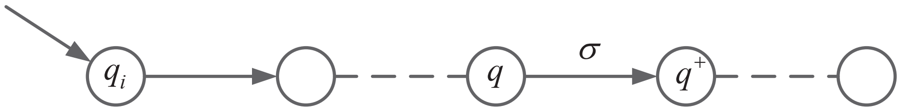


Figure 3.1: An automaton.

States

A state is the condition or situation of a system with regard to certain rules, policies and physical laws that are applied to the system. In Fig. 3.1, q and q^+ are two states of the system.

Events

An event happens at a time instant which causes a change in the state of the system (or staying in the current state). The events are atomic and cannot be interrupted. Regarding the particular event-sensors used, some of the events are observed by the interacting systems. Such a partial observation partitions the events into two types of event;either they are *observable* or *unobservable*.

Observable events are trackable by the interacting system in a specific state and it means that there is a sensor to record the state transition. The observable events are typically commands issued from the controller, sensor recordings after the execution of the controller commands, and also changes in the recordings of the sensors [11].

Unobservable events are not recordable by the sensors and are not trackable. Failure events which do not cause any immediate change in the sensor readings and silent events are examples of unobservable events. Silent events, may cause a change in the state of the system but are not observable by an outside observer.

Transition Function and Transition Relation

As illustrated in the Fig. 3.1, a transition function δ is $q^+ = \delta(q,\sigma)$, which means that for a state $q \in Q$ and an event $\sigma \in \Sigma$, the next state is $q^+ \in Q$. Moreover,

$$q \xrightarrow{\sigma} q^+$$

denotes the relation $T = \langle q,\sigma,q^+ \rangle$.

Coding States by Integers and Transitions by Predicates

In (3.1) Q is the finite set of states, and can be coded by integers to obtain a more compact model of the system. Considering that each state q is encoded by one or more variables v, (3.1) can be rewritten as

$$A = \langle X,\Sigma,T,v_0 \rangle \tag{3.2}$$

where

(i) $\langle x_1, \ldots, x_n \rangle \in X_1 \times \cdots \times X_n$, where x is a state vector and X_i is the domain of the state variable x_i,

(ii) $\Sigma = \{\sigma_1, \sigma_2, \ldots, \sigma_p\}$ is the finite set of events,

(iii) $T : X \times \Sigma \times X \to \mathbb{B}$ is the transition predicate, and

(iv) x_0 is the initial state.

Note that both the notions transition relation $T : V \times \Sigma \times V$ and transition predicate $T : V \times \Sigma \times V \to \mathbb{B}$ are used in the text interchangeably. As mentioned before, the focus of this thesis is on nondeterministic EFAs, where the nondeterministic part needs utilizing transition relations, and the EFA part needs to define conditions which are predicates on the variables.

Example 3.1

Figure 3.2 represents a buffer, where each of its states are coded by two variables, $v = \langle v_1, v_2 \rangle$, where $v_1 \in \{0,1,2\}$ and $v_2 \in \{0,1\}$.

This form of representation by coding enables us to have an equation-based model or a logical model of the system that is more closely related to the evrification tool NuSMV, which is used in this thesis for verification of diagnosability. The logical model of Fig. 3.2 is

Event a**:** $v_1^+ = v_1 + 1 \quad \text{if } v_1 < 2.$

Event b**:** $\begin{cases} v_1^+ = v_1 - 1 \\ v_2^+ = v_2 + 1 \end{cases} \quad \text{if } v_1 > 0,\ v_2 < 1.$

Event c**:** $v_2^+ = v_2 - 1 \quad \text{if } v_2 > 0.$

The above logical model of transitions is also coded by predicates in the following. The predicate representation is used more in the next chapters.

$$\begin{aligned} &T_a : \sigma = a \land v_1 < 2 \land v_1^+ = v_1 + 1 \\ &T_b : \sigma = b \land v_1 > 0 \land v_2 < 1 \land v_1^+ = v_1 - 1 \land v_2^+ = v_2 + 1 \\ &T_c : \sigma = c \land v_2 > 0 \land v_2^+ = v_2 - 1 \end{aligned}$$

Nondeterministic Automata

A discrete event system can be modeled as a deterministic automaton as in Fig. 3.3(a), where only one outgoing transition is enabled for each event. Since there is a unique outgoing transition for each event, the transition relation for deterministic automata is also a function.

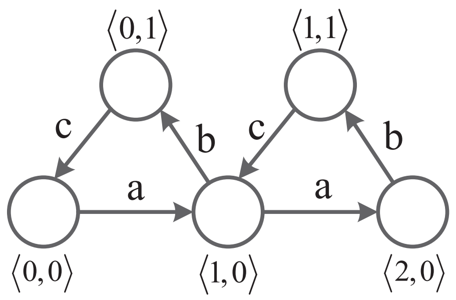
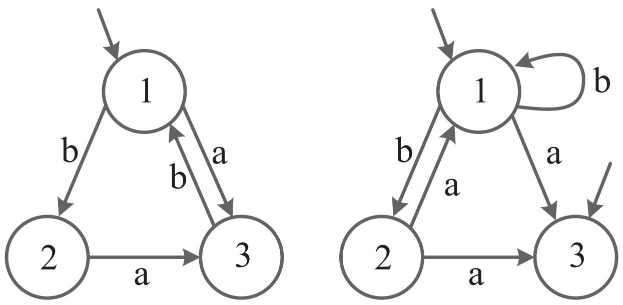


Figure 3.2: A buffer example.



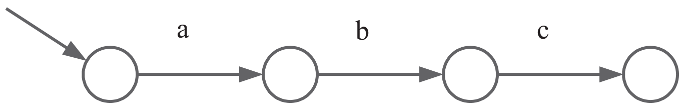
(a) A deterministic automaton (b) A nondeterministic automaton

Figure 3.3: An example of (a) a deterministic and (b) a nondeterministic automaton.

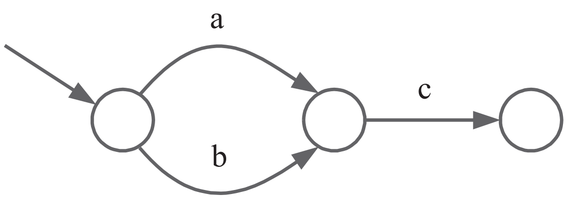
On the other hand, a nondeterministic automaton allows multiple outgoing transitions for a single event, starting from a specific state as in Fig. 3.3(b). Therefore the transition relation is $T : V \times \Sigma \times 2^V$, where 2^V is the power set of V, which is the set of all subsets of V, e.g., $1 \xrightarrow{b} \{1,2\}$ as in Fig. 3.3(b). Moreover, a nondeterministic automaton may have more than one initial state which can nondeterministically be chosen. As it is shown in the Fig. 3.3(b), both states 1 and 3 are initial states.

3.3 Formal Languages

As described before, the event set Σ is an alphabet and sequences of events of the alphabet form *words* or *strings* of events. In the literature, the term *trace* is also used. If there is a string with no events, it is called an empty string and is shown by ε. The *Language* of an automaton is the set of all traces that can be executed and is represented by $L(A)$. The number of events in a string, indicates the length of the string, counting the multiple occurrences of the same event, and it is shown by $|s|$, where s is a string.



(a) $L(A) = \{\varepsilon, a, ab, abc\}$



(b) $L(A) = \{\varepsilon, a, b, ac, bc\}$

Figure 3.4: Two Simple automata.

Example 3.2

Let $\Sigma = \{a,b,c\}$ be the set of events. The language of A can have different elements based on the structure of the automaton. In Fig. 3.4(a) the language is $L(A) = \{\varepsilon, a, ab, abc\}$, while in Fig. 3.4(b) we have $L(A) = \{\varepsilon, a, b, ac, bc\}$ representing an alternative choice between the event a or b.

Set of All Strings

The set of all strings is constructed by concatenating the alphabet of the automaton including the empty string ε and is shown by Σ^*. Thus, *Concatenation* is to build strings and languages from an event set Σ. The set of all strings for both automata in Fig. 3.4 is $\Sigma^* = \{\varepsilon, a, b, c, aa, ab, ac, ba, bb, bc, ca, cb, cc, aaa, aab, aac, aba, abb, abc, \ldots\}$. Note that Σ^* is infinite.

Prefix closure

Before explaining the prefix closure operation on languages, it is better to define some terminologies on strings. They are described as follows. Consider there is a string $abc = s$, where a, b and c are also strings that belong to Σ.

- a is a prefix of s,
- a, b and c are substrings of s, and
- c is a suffix of s.

Now, consider $L \subseteq \Sigma^*$, then the prefix closure of the language L is denoted by $\bar{L}$ and consists all of the prefixes all of strings in L.

$$\bar{L} = \{s \in \Sigma^* : (\exists a \in \Sigma^*)\, [sa \in L]\} \tag{3.3}$$

In general, $L \subseteq \bar{L}$, but if $L = \bar{L}$, L is called *prefix closed*.

Projection of Strings

Projection of a string is denoted by the symbol P, and it is a mapping from a set Σ_l to a smaller set of events Σ_s, where $\Sigma_s \subset \Sigma_l$. Thus it is

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

where

$$\begin{aligned} &P(\varepsilon) := \varepsilon \\ &P(\sigma) := \begin{cases} \sigma & if\ \sigma \in \Sigma_s \\ \varepsilon & if\ \sigma \in \Sigma_l \backslash \Sigma_s \end{cases} \\ &P(s\sigma) := P(s)P(\sigma) \quad for\ s \in \Sigma_l^*, \sigma \in \Sigma_l \end{aligned} \tag{3.4}$$

Example 3.3

Consider the event set $\Sigma_l = \{a,b,f\}$ with a and b as observable events and f as a failure event which is an unobservable event. The projection P is defined as an observer (defined in the next chapter), which maps the event set to the observable event set $\Sigma_s = \{a,b\}$. Take the language as

$$L = \{f,af,ffb,fab,afb,fabfba\} \subset \Sigma_l^*. \tag{3.5}$$

Since the projection of this language $P(L)$ is generated by taking the projection of the involved strings in L, we obtain

$$P(L) = \{\varepsilon,a,b,ab,abba\}.$$

Inverse Projection

The inverse projection $P^{-1} : \Sigma_s^* \to 2^{\Sigma_s^*}$ is defined as follows

$$P^{-1}(t) := \{s \in \Sigma_l^* : P(s) = t\}$$

where $2^{\Sigma_s^*}$ denotes the power set of Σ_s^* which is the set of all subsets of Σ_s^*. The extension of the projection and the inverse projection on the languages are

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

and for $L_s \subseteq \Sigma_s^*$

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

Note that in general, $P^{-1}[P(L)] \neq L$ for a given language $L \subseteq \Sigma_l^*$.

Example 3.4

Consider the event sets Σ_s and Σ_ℓ in example 3.3 and the language in (3.5). Here are some examples on the inverse projection.

$$\begin{aligned} P^{-1}(\{\varepsilon\}) &= \{f\}^* \\ P^{-1}(\{a\}) &= \{f\}^*\{a\}\{f\}^* \\ P^{-1}(\{bba\}) &= \{f\}^*\{b\}\{f\}^*\{b\}\{f\}^*\{a\}\{f\}^*. \end{aligned} \tag{3.6}$$

Once again note that $P(P^{-1}(L)) = L$, while $P^{-1}(P(L)) \neq L$.

3.4 Synchronous Composition

Synchronous composition is the interaction between two automata with the events in their alphabets. Consider $A = \langle Q^A, \Sigma^A, T^A, q_i^A \rangle$ and $B = \langle Q^B, \Sigma^B, T^B, q_i^B \rangle$ as the two automata where we are interested in their interaction. Their synchronous composition is

$$A \parallel B = \left\langle Q^A \times Q^B, \Sigma^A \cup \Sigma^B, T, \left\langle q_i^A, q_i^B \right\rangle, Q_m^A \times Q_m^B \right\rangle \tag{3.7}$$

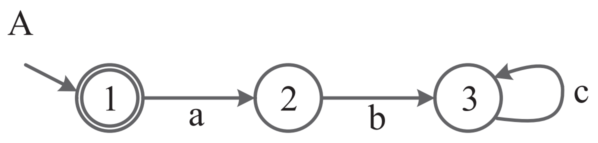
with the transition relation T defined as

$$(\left\langle q^{(A\|B)} \right\rangle \overset{\sigma}{\to} \left\langle q^{(A\|B)+} \right\rangle) = \begin{cases} (q^A \overset{\sigma}{\to} q^{A+}) \times (q^B \overset{\sigma}{\to} q^{B+}) & \sigma \in \Sigma^A \cap \Sigma^B \\ (q^A \overset{\sigma}{\to} q^{A+}) \times \{q^B\} & \sigma \in \Sigma^A \backslash \Sigma^B \\ \{q^A\} \times (q^B \overset{\sigma}{\to} q^{B+}) & \sigma \in \Sigma^B \backslash \Sigma^A \end{cases} \tag{3.8}$$

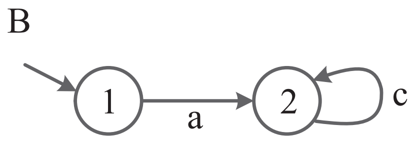
where q_i is the initial state and Q_m is a marked state. Marked state is a desired state that must be reachable. Typically a marked state is a state where a task has been completed.

Example 3.5

In Fig. 3.5 the synchronous composition between two automata A and B is shown for the two cases when $\Sigma^B = \Sigma^A = \{a,b,c\}$ and $\Sigma^B = \{a,c\}$. The automaton B is given without marked states, which means that both states are assumed to be marked. The cross product $Q_m^A \times Q_m^B = \{1\} \times \{1,2\} = \{\langle 1,1 \rangle, \langle 1,2 \rangle\}$, where only the first state $\langle 1,1 \rangle$ is reachable. For simplicity, the brackets <> are excluded in the notations of the states in Fig. 3.6. As can be seen in Fig. 3.6(a), only the event a can be executed in both automata. Then, since event b is in the language of both automata, and it is blocked (prevented from executing) in automaton B, it is blocked in their synchronization as well. On the other hand, for the case that $b \notin \Sigma^B$, all events are allowed to be executed, as shown in Fig. 3.6(b).

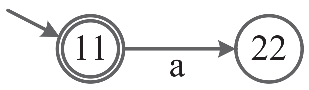


(a) Automaton A

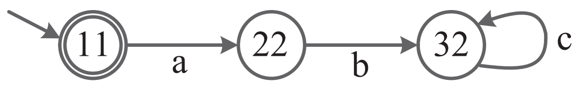


(b) Automaton B

Figure 3.5: Automata A and B which are considered for synchronization.



(a) $b \in \Sigma^B$



(b) $b \notin \Sigma^B$

Figure 3.6: The synchronous composition between the automata A and B with two different alphabets for B, where in (a) $b \in \Sigma^B$ and (b) $b \notin \Sigma^B$.

3.5 Conclusion

In this chapter the automaton was presented as a modeling formalism for DES. Moreover, the coding of states by integers and transitions by predicates was briefly introduced. This modeling formalism will be used in the next chapters. The coding of states by integers makes it easier to introduce states, or locations in EFAs, as integer sets. Furthermore, in the NuSMV implementation this coding by integers and predicates, make the implementation easier and reduces the complexity.

Nondeterminism in automata was introduced in this chapter, which this concept is added also to the EFAs in the following chapters. Furthermore, the interaction of two automata was introduced as the synchronous composition. In the next chapter, a projected synchronous composition is also introduced which is a part of diagnosability test algorithm and is different from the normal synchronization from the unobservable event synchronization aspect. Formal languages are also introduced, and some specific notions, specially the projection and its inverse. These notions are important when diagnosers are defined.

4

Diagnosability and Failure Diagnosis

4.1 Introduction

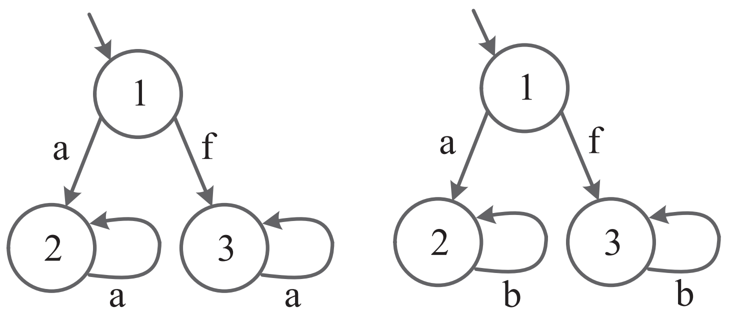
In many applications where the system contains unobservable events, we may be interested in determining whether certain unobservable events have occurred in the system. This is the problem of event diagnosis. If multiple failures need to be diagnosed, we can either build one diagnoser for each failure or build a single diagnoser that simultaneously tracks all failure events of interest.

In this chapter the concept of diagnosability is explained and two algorithms are presented for testing the diagnosability property. These algorithms are based on the concept of indeterminate cycles, which is also described in this chapter. Moreover, after verifying the diagnosability of the system, an online diagnosis algorithm is illustrated, which can diagnose failures by tracking the reachable states of the system. This algorithm can also be used for non-diagnosable automata where some but not all the failures are diagnosable.

Note that the diagnosability test is possible by verifying the states of the system, which can be done by a model-checker. A model-checker can verify the diagnosability of the system by checking the considered specific specification. In this chapter, two CTL specifications are proposed for diagnosability test. This is closely related to a verification procedure suggested in [5].

Diagnosability

To start this section, it is fruitful to explain diagnosability by two simple examples as shown in Fig. 4.1. The events a and b are observable while f is an unobservable event that is also a failure. In Fig. 4.1(a), observing event a, it is not clear whether the transition containing a failure is executed or not, since both transitions have the same language $\{\varepsilon, a, aa, aaa, \ldots\}$. Thus, the failure occurrence can not be distin-



(a) Non-diagnosable automaton (b) Diagnosable automaton

Figure 4.1: Diagnosability in automata

guished and therefore, the automaton is non-diagnosable. On the other hand, in Fig. 4.1(b), two different languages (sets of strings) starting from the initial state can be distinguished, which are $\{\varepsilon,a,ab,abb,abbb,\ldots\}$ and $\{\varepsilon,b,bb,bbb,\ldots\}$. Therefore, observing the two different languages, one can distinguish if a failure has happened. In other words, as soon as the event b is observed without observing a before, we know that the failure f has happened.

Diagnosability is the ability to deduce about the past occurrence of unobservable failure events from a bounded number of observable events. In [5], an event observation projection for diagnosis is defined where the event set is mapped to a smaller event set only containing observable events. Therefore, after projecting the event set, only observable events are seen, and the automaton can be interpreted as an observer. In [3], this method is called passive diagnosis, while the integrated method where observation is integrated in a control strategy is called active diagnosis.

Definition 4.1- Event observation projection

In passive diagnosis, the event observation projection, called observation mask in [5], is a mapping from the original event set Σ to the smaller observable event set $\Sigma_o \subseteq \Sigma$, i.e., $P : \sum \rightarrow \sum_o \cup \{\varepsilon\}$ that can be extended to $\sum^*$, so we have $s \in \sum^*$, $\sigma \in \sum$: $P(s\sigma) = P(s)\,P(\sigma)$, with $P(\varepsilon) = \varepsilon$ and $P(\sigma) = \varepsilon$ for all $\sigma \in \sum$. Here, $\sum$ denotes an unobservable event set.

Considering the language of the system to be live[1], without any cycle of unobservable events, the notion of diagnosability is as follows; if s is a trace in the language of automaton $L(A)$ ending with an F_i type failure, and m is a sufficiently long trace obtained by extending s, then every trace w that is observation equivalent to m , i.e., $P(w) = P(m)$, should contain an F_i type failure. It is formulated in Definition 4.2.

[1]The liveness assumption is made for simplicity. Live languages are normally defined as the languages which do not have terminating strings, or in other words, have no deadlock [1].

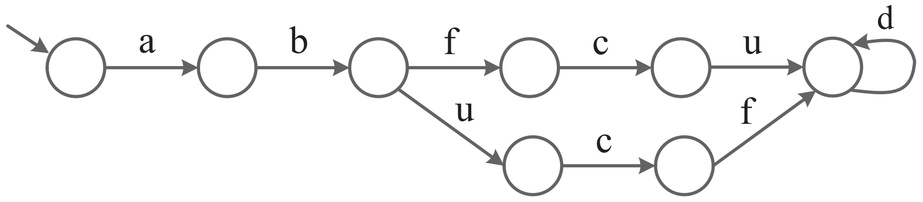


Figure 4.2: Diagnosability in automata

Failure Assignment Function

Failure assignment function is a mapping from the original event set Σ to either 0 or $\mathcal{F}$.

$$\psi : \Sigma \to \mathcal{F} \cup \{0\}$$

where $\mathcal{F} = \{F_i, i = 1, \ldots, m\}$. It means that if $\sigma \in \Sigma$ is not a failure event it is projected to 0. Otherwise it is projected to the F_i type failure set where it belongs to. For instance, in Fig. 4.1(a), $\Sigma = \{a, f\}$ where a is not a failure, and it is projected to 0, while f is a failure event and it is projected to $\mathcal{F}$.

Definition 4.2- Diagnosability

With respect to the event observation projection introduced in Definition 4.1 and the failure assignment function $\psi : \Sigma \to \mathcal{F} \cup \{0\}$, a prefix-closed language L is diagnosable if the following formula holds.

$$\begin{aligned}&(\forall F_i \in \mathcal{F})(\exists n_i \in \mathbb{N})(\forall s \in L, \psi(s_f) = F_i)(\forall m = st \in L, \|t\| \geq n_i)\\ &\Rightarrow (\forall w \in L, P(w) = P(m))(\exists r \in pr(\{w\}), \psi(r_f) = F_i) \qquad (4.1)\end{aligned}$$

Here, s_f and r_f are the last events in traces s and r respectively, $pr(\{w\})$ is the set of all prefixes of w. A system is called diagnosable if its language L is diagnosable.

Example 4.1

In Fig. 4.2, consider the upper trace as $m = st$, where $s = abf$ with the last event as failure f, and $t = cud$ which is the extension of s and includes u as an unobservable event. The projection P over the trace is $P(m) = abcd$. There is only one additional trace $w = abucfd$, that has the same projection as m, i.e., $P(m) = P(w) = abcd$. Considering the set of prefixes of w, $pr\{w\} = \{\varepsilon, a, ab, abu, abuc, abucf, abucfd\}$, it is seen that there exists a trace $r = abucf$, which ends in a failure event. Therefore, following Definition 4.2, the automaton depicted in Fig. 4.2 is diagnosable.

Observer

The observer is an automaton that is constructed based on the original automaton, where the states of the observer represent the set of possible states the system can be in after the execution of a trace of observable events. An observer has no knowledge about the executed unobservable events in the sequence of events executed by the system. The diagnoser is a refined type of observer.

4.2 Diagnoser

Offline verification of the diagnosability property of a system is a demand to get a diagnoser. Moreover, online detection and isolation of the failure while the system is operating is the main purpose for implementing the diagnoser [1].

Offline Diagnosis

When the whole system, i.e. states and events including failures are considered to perform the diagnosability test algorithm, the method is called offline diagnosis. In this case, after implementing a projection, an observer automaton with added failure states is obtained as a diagnoser. Offline diagnoser is different from online diagnoser from the implementation point of view.

In the sequel, there is a failure label F which is augmented to each state of a diagnoser; $F = 0$ means σ of the ingoing transition is not a failure event and also means no failure has occurred so far, whereas $F = 1$ implies that a failure has occurred. As soon as F becomes 1, it keeps its value for the rest of the trace. For example, in Fig. 4.1(a) and 4.1(b), state 3 can be augmented with $F = 1$, which implies that the ingoing transition contains a failure.

Online Diagnosis

In this case, the failure diagnosis is performed by tracking the current state of the diagnoser in response to the observable events executed by the system. Implementing this technique, it is guaranteed to detect every failure within a bounded delay after the occurrence, indeed, after checking that the diagnosability test holds. However, even if the diagnosability test does not hold, the diagnoser may diagnose some of the failure types [5].

The difference between offline and online diagnoser is that in an online version the whole diagnoser is not generated, the projection is generated only for the specific trace that is generated by the system.

Reduced Diagnoser

This diagnoser is constructed by removing any state of the offline diagnoser that is either impossible or where the diagnoser is certain that a fault has previously occurred. Therefore, any feasible sequence that is not executable in the reduced diagnoser automaton indicates that a failure must have occurred in the past [5].

Decentralized Diagnosis

Assume that the system has a set of observable events where in each local diagnoser only some of them are observable. Generally, this means that each diagnoser can diagnose one or more failures based on its observed event trace. To work properly, each failure must be diagnosed in at least one diagnoser, and the other diagnosers would remain in indeterminate cycle, which is explained in the following section, upon the occurrence of the string containing that failure. This method is also called *codiagnosability* [9].

4.3 Uncertain and Indeterminate Cycles

Uncertain Cycle

Consider automaton G_1 and its diagnoser D_1 in Fig. 4.3 with f as failure event and F as a label to indicate failure occurrence by being equal to 1, otherwise 0. The associated labels are propagated with states following certain rules for label propagation which indicates that as soon as F becomes equal to 1, it keeps its value. As it is seen in D_1, there is a loop with a single event a in the initial state which includes different failure labels. This state is called uncertain state. Generally, an uncertain state in a diagnoser includes states of corresponding system carrying both $F = 0$ and $F = 1$ labels. Uncertain cycle is a loop where all states are uncertain.

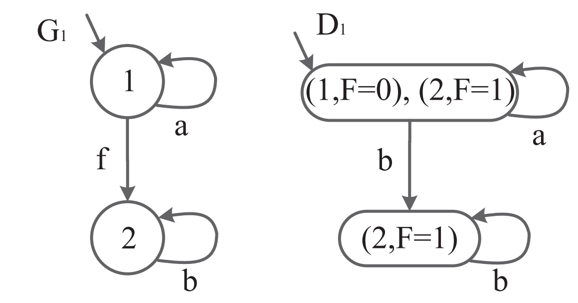
It can be seen in Fig. 4.3 that the diagnoser D_1 only contains one uncertain cycle corresponding to one cycle in G_1 with label $F = 0$. Thus automaton G_1 is diagnosable since there is no other a loop in G_1 including $F = 1$. Therefore, traversing in loop a in D_1 does not violate the ability to diagnose the failure because there is no ambiguity in inverse projection from loop a in D_1 to loop(s) a in G_1.

In other words, to recognize whether an automaton is diagnosable or not, it is necessary to consider both the system and its diagnoser. First, find an uncertain cycle in diagnoser D_1 as in Fig. 4.3. Then, check corresponding cycles in the system G_1 which are the inverse projection of that cycle in D_1. If there is a cycle in G_1 merely with either labels $F = 0$ or $F = 1$, the corresponding cycle in D_1 is called uncertain cycle, and the system is diagnosable. In Fig. 4.4 the diagnoser depicted in Fig. 4.4(b) does not contain any uncertain cycle and the system depicted in Fig. 4.4(a) is diagnosable.

Otherwise, if the inverse projection from D_2 to G_2, results in two cycles, one containing $F = 0$ and the other one $F = 1$, the cycle in the diagnoser D_2 is called indeterminate cycle and the automaton is non-diagnosable, which is described in the following.

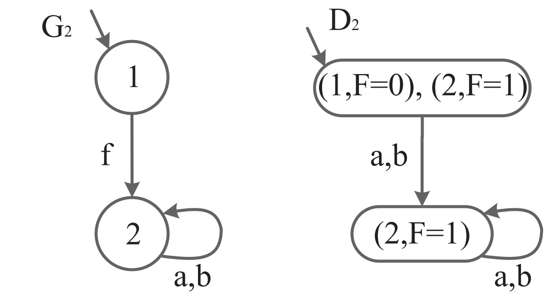
Indeterminate Cycle

In general, a cycle in a diagnoser is called an indeterminate cycle, if two cycles in the system automaton G, one with label $F = 0$ and the other one with $F = 1$, can be



(a) The diagnosable system G_1 (b) The diagnoser D_1

Figure 4.3: Example of a system with a cycle of uncertain states in its diagnoser D_1.



(a) The diagnosable automaton G_2 (b) The diagnoser D_2

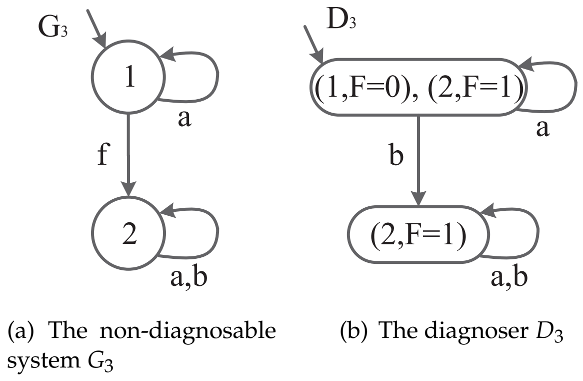
Figure 4.4: Example of a system with no cycle of uncertain states in its diagnoser D_2.

associated with a cycle of uncertain states in D. By definition, the presence of an indeterminate cycle implies a violation of diagnosability. The diagnoser in Fig. 4.5(b), has one uncertain cycle a which corresponds to two similar cycles in G_3 depicted in Fig. 4.5(a). As mentioned above, there is an ambiguity in projecting back from the uncertain cycle in D_3 to the corresponding cycles in G_3 and the diagnoser can not distinguish whether the system is cycling in the state 1 or 2. Therefore, the diagnoser contains an indeterminate cycle, and the system is not diagnosable.

Consequently, searching for any cycle of uncertain states in the diagnoser, and checking whether it is indeterminate, is a handy method to find out the diagnosability of automata before performing any further task. The algorithms explained in Section 4.5 are based on this concept.

Remark

Consider Fig. 4.6 and 4.7, where both of them contain an uncertain cycle b in the diagnoser which is depicted in Fig. 4.6(b) and 4.7(b) observing identical diagnosers.



(a) The non-diagnosable system G_3

(b) The diagnoser D_3

Figure 4.5: Example of a system with an indeterminate cycle in its diagnoser D_3.

By looking solely at the diagnosers one may conclude that both systems have the same behavior and both are diagnosable because there is an exiting transition c from the uncertain state which leads to a certain state with label $F = 1$. But this is wrong, since the diagnoser depicted in Fig. 4.6(b) has uncertain states and the system is diagnosable while the diagnoser in Fig. 4.7(b) includes indeterminate cycle and the system is non-diagnosable.

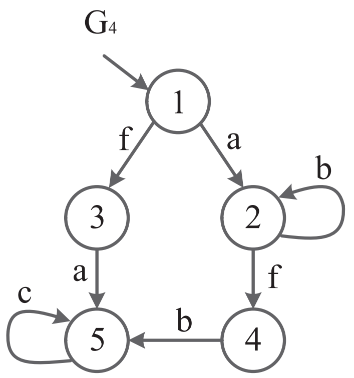
These examples show the importance of considering both diagnoser and system together; although diagnosers may be identical but their corresponding systems may behave differently.

4.4 Temporal Logic

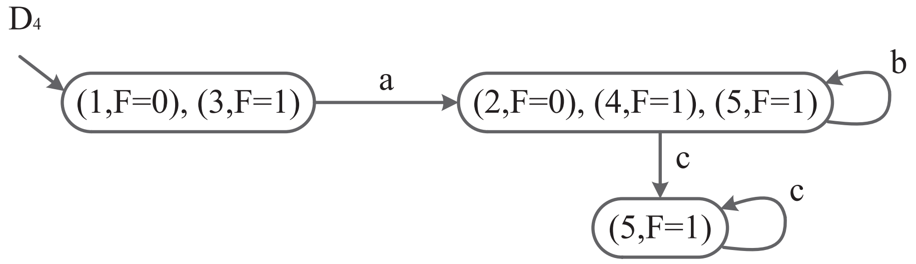
To be able to apply an online diagnoser for a discrete event system, it is necessary to know that the system is diagnosable. One of the approaches that is used to verify this property is the model-based approach, in which the system and the specification are represented by a model $\mathcal{M}$ for an appropriate logic and a formula Φ, respectively. Model-checking is based on temporal logic. The verification method checks whether $\mathcal{M}$ satisfies Φ, (written $\mathcal{M} \vDash \Phi$). This can be done automatically for finite state models.

There are particularly two often used logics, the *linear-temporal logic* (LTL) where time is linear, and the *computation-tree logic* (CTL) where time is branching. Temporal logic implements temporal quantifiers, which in CTL are expressed in pairs. In CTL, as well as the temporal operators U, F, G and X of LTL there are also quantifiers A (universal quantifier) and E (existential quantifier) which express "all paths" and "exists a path", respectively [12]. Furthermore, G is the universal quantifier and F is the existential quantifiers, *ranging over the states along a particular path*. Moreover, X is read as next and U is read as until.

To implement the diagnosability test algorithm for finding out whether there exist indeterminate cycles in $x_i = ((x_i^1,f_i^1),(x_i^2,f_i^2))$, $i = 1, 2, \ldots, n$, two CTL model



(a) The diagnosable system G_4



(b) The diagnoser D_4

Figure 4.6: Example of a system with a cycle of uncertain states in its diagnoser D_4.

checking formulas are used as specification. The formulas are related to each other based on de Morgan rules.

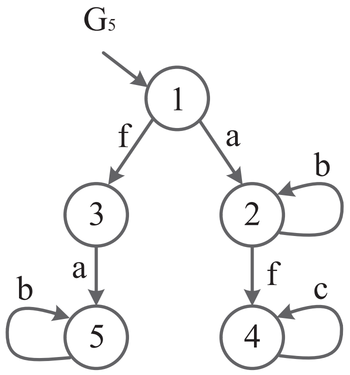
$$\begin{aligned} \neg EF(\Phi) &\equiv AG(\neg\Phi) \\ \neg AF(\Phi) &\equiv EG(\neg\Phi). \end{aligned} \tag{4.2}$$

Based on (4.2), the two formulas $EFEG(\Phi)$ and $AGAF(\Phi)$ are the negation of each other, which is shown in the following equivalences.

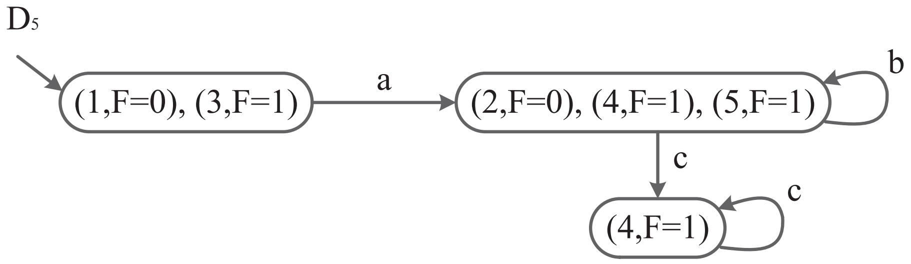
$$EFEG(\neg\Phi) \equiv EF(\neg AF\Phi) \equiv \neg AGAF\Phi. \tag{4.3}$$

Assume that f_i^1 and f_i^2 are the failure labels that are augmented to each state of the automaton and its copy, respectively.

When $EF\,EG\ (f_i^1 \neq f_i^2)$ becomes true, it means that there is an indeterminate cycle, and then the system is non-diagnosable. It means that the negation $AG\,AF\ (f_i^1 = f_i^2)$ is true when the system is diagnosable. These CTL specifications are now further mentioned.



(a) The non-diagnosable system G_5



(b) the diagnoser D_5

Figure 4.7: Example of a system with an indeterminate cycle in its diagnoser D_5.

CTL Specifications

Let $\mathcal{M} = (X, \rightarrow, L)$ be a model for CTL, where x in X a state of the model, Φ a CTL formula and L the language. The relation $\mathcal{M},x \vDash \Phi$ is defined by structural induction in Φ. Here, the two relations that are used for diagnosability verification are described:

1. $\mathcal{M},x \vDash EF\, EG\, \Phi$, where $\Phi : f_i^1 \neq f_i^2$, holds iff there is a path $x_1 \rightarrow x_2 \rightarrow x_3 \rightarrow \cdots$, where x_1 equals x, and for at least one trace starting from x, we have $\mathcal{M},x_i \vDash \Phi$ for all $i \geq k$. Mnemonically: there exists a computation path initializing in x such that Φ holds globally in all future states along at least one path after a limited number of states have been passed or in other words, in at least one path Φ will eventually be permanently true.

   Figure 4.8 indicates that Φ is not true in the initial state, according to the concept that Φ conveys. Moreover, it illustrates a "minimal" way of satisfying the formula, i.e., the existence of only one trace is enough for the specification to be evaluated as true. Φ can switch between true and false but, after a limited number of steps it must always be true in at least one trace.

2. $\mathcal{M},x \vDash AG\, AF\, \Phi$, where $\Phi : f_i^1 = f_i^2$, holds iff for all paths $x_1 \rightarrow x_2 \rightarrow x_3 \rightarrow$

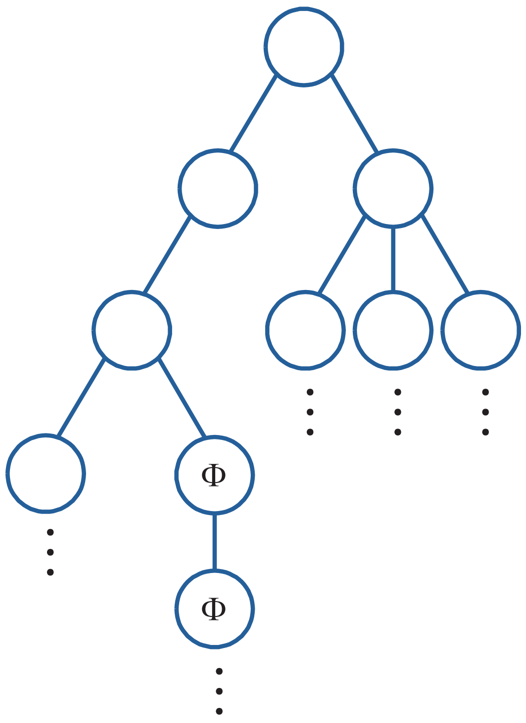


Figure 4.8: A system whose starting state satisfies EF EG Φ. The states marked with Φ are evaluated as true.

$\cdots$, where x_1 equals x, and all x_i along the path, we have $\mathcal{M},x_i \models \Phi$ for all $i \geq k$. Mnemonically: for all computation paths beginning from x, there will be some future states where Φ holds infinitely often. Note that "along the path" includes the path's initial state.

4.5 Offline Diagnosis Algorithms

Diagnosability Test Algorithm 1

In [4], a polynomial algorithm is proposed to perform diagnosability test without constructing a diagnoser, in contrast to the approach introduced in [1] which tests diagnosability by constructing the diagnoser resulting in exponential complexity. The polynomial algorithm is as follows:

1. Refine the state set Q by augmenting the mapped states using P with the set of failure types along certain paths from q_0 to q, using the failure assignment function ψ to obtain a nondeterministic finite state automaton A_o. $Q_o = \{(q,f)|q \in Q_1 \cup \{q_o\}, f \subseteq \mathcal{F}\}$ is the finite set of states, where $Q_1 = \{q^+ \in Q|\exists(q,\sigma,q^+) \in T$ with $P(\sigma) \neq \varepsilon\}$ are the states of A that are reachable by an observable transition.

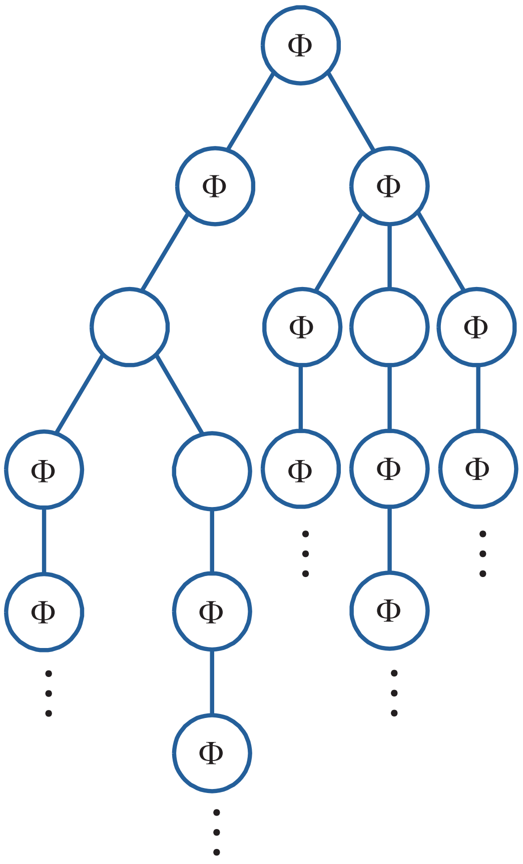
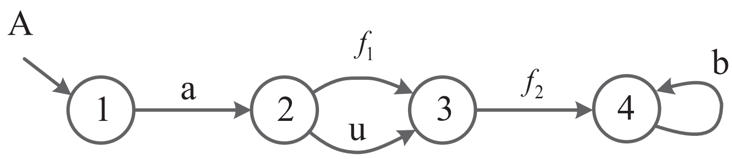


Figure 4.9: A system whose starting state satisfies AG AF Φ. The states marked with Φ are evaluated as true.

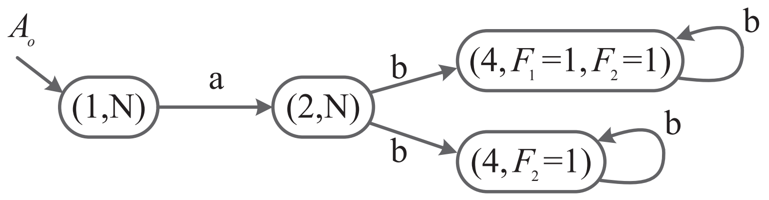
2. Compute the strict composition of the automaton A_o with itself, $A_d = (A_o \parallel A_o)$. The steps 1 and 2 generate the *projected synchronous composition* of two copies of the refined A.

3. Check A_d to find whether there exists, a cycle of states, $q_i = ((q_i^1, f_i^1), (q_i^2, f_i^2))$, $i = 1, 2, \ldots, n$, where the failure types assigned to each state of A_d (q_i^1 and q_i^2) are different, i.e, $f_i^1 \neq f_i^2$. Then the automaton is not diagnosable. Otherwise, the automaton is diagnosable.

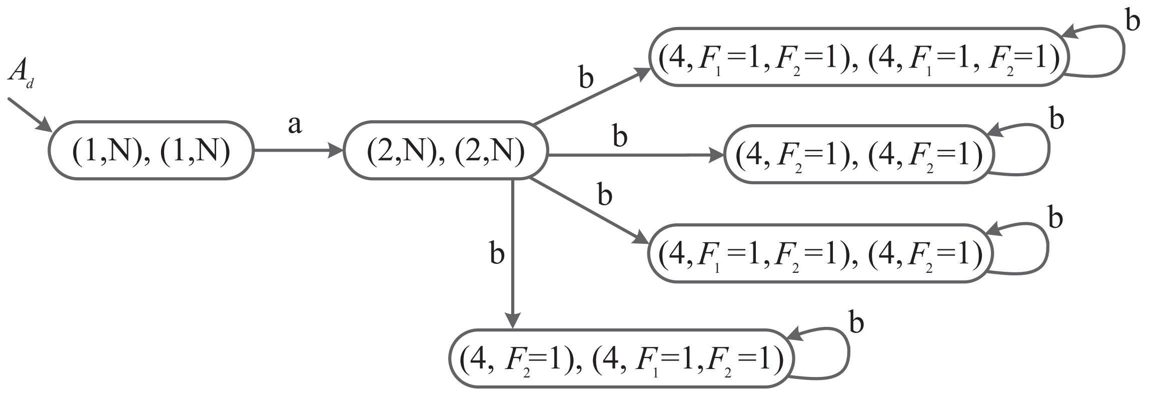
In summary, this method checks the diagnosability of the system only by checking each state of A_d, to find out whether it has any cycle containing two non-identical failure types. In the case that such cycles can be found, the system is reported as non-diagnosable. The last step can be performed in another way; deleting all q_i states in A_d except states that include two different f_i^1 and f_i^2 labels, and check whether the remainder graph contains a cycle or not. This method originates from the concept of indeterminate cycles, which is explained in Section 4.3.



(a) The system automaton A.



(b) The diagram of A_o.


A_d
(1,N), (1,N)
a
(2,N), (2,N)
b
$(4, F_1{=}1, F_2{=}1)$, $(4, F_1{=}1, F_2{=}1)$
b
b
$(4, F_2{=}1)$, $(4, F_2{=}1)$
b
b
$(4, F_1{=}1, F_2{=}1)$, $(4, F_2{=}1)$
b
b
$(4, F_2{=}1)$, $(4, F_1{=}1, F_2{=}1)$
b

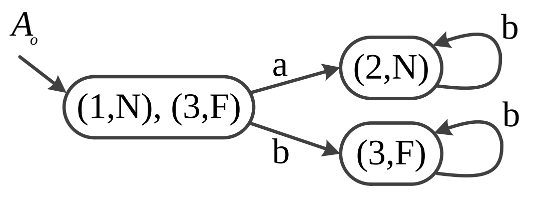
(c) The diagram of A_d.

Figure 4.10: Diagnosability test approach where (a) is the system automaton, (b) is the refined observer, and (c) is the automaton which is resulted from $A_d = (A_o \parallel A_o)$.

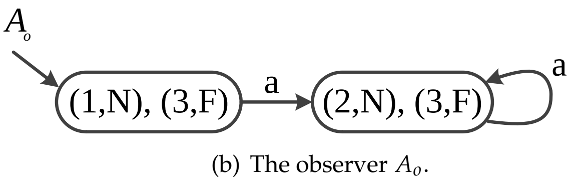
Note that this test gives an overall insight about the diagnosability of the system. In other words, it does not show the event trace that has led to a failure but merely checks whether the system is diagnosable.

Example 4.2

Consider the automaton in Fig. 4.10(a). From the first step of the algorithm, A_o can be derived from A, which is shown in Fig. 4.10(b). Note that here N denotes that no failure has happened in both automata. Figure 4.10(c) shows the strict composition of A_o with itself $A_d = (A_o \parallel A_o)$ which is described in step 2 in the algorithm. Performing step 3 in the algorithm, there are self loops where the labels in each state are not the same. Hence, the automaton is not diagnosable. In case $F_1 = F_2$ in Fig. 4.10(c) and deleting the redundant states, the resulting automaton is diagnosable.



(a) The observer A_o.



(b) The observer A_o.

Figure 4.11: Diagnosability test approach where (a) is the refined observer of the automaton in Fig. 4.1(a), and (b) is the refined observer of the automaton in Fig. 4.1(b).

Example 4.3

In this example, the diagnosability test is implemented on the two automata depicted in Fig. 4.1. Here, only A_o of both automata are depicted, because in both automata the structure of A_o and A_d are the same. As it is seen in Fig. 4.11(a), the automaton has two loops on non-ambiguous states, which is also the case in the synchronized automaton A_d. Therefore, the automaton in Fig. 4.11(a), is diagnosable. However, the automaton which its A_o is depicted in Fig. 4.11(b), has a loop on an ambiguous state in A_d as well. Thus, it is non-diagnosable.

Diagnosability Test Algorithm 2 (Rule-Based Model)

This method introduced in [5] is the generalization of the diagnosability test presented in [4], which was described in the previous algorithm, since it is a more computational oriented approach. In the same as in algorithm 1, it is assumed that there is no cycle of unobservable events in A and a binary valued variable F is presented which indicates whether a failure occurred in the past or not. The state transition is presented using a set of rules, which is also called a guard relation in this rule-based model. It is as follows

$$T_\sigma : G_\sigma(q) \Rightarrow q \xrightarrow{\sigma} q^+ \tag{4.4}$$

where $\sigma \in \Sigma$ is an event, $G_\sigma(q)$ is the enabling condition predicate or the guard. The above statement is enabled if $G_\sigma(q)$ holds, then the state variables may be updated to the new values and a state transition occurs. F is equal 1 when the failure event occurs and once it becomes 1, it remains at this value. Moreover, a faulty-state predicate is also defined as $B(q) = B((v,F)) := [F = 1]$. With this $B(q)$ and the extended rule-based predicate, the algorithm is described in the following

1. Augment the state set variables with F so that the state variable will be $q = (v,F)$. Then extend the rule-based model to include the new q. Thus, the initial state is given by the predicate $I(q) := I(v) \wedge [F = 0]$

2. Perform the *projected synchronous composition* of augmented A with its copy. For the copy of A use variable p instead of q. Considering the event occurrence rule,

   $$\forall(\sigma,\sigma') \in [(\Sigma_\varepsilon \times \Sigma_\varepsilon) - \{\varepsilon,\varepsilon\}], \text{ s.t. } P(\sigma) = P(\sigma')$$

   where $\Sigma_\varepsilon = \Sigma \cup \{\varepsilon\}$, the synchronization is as follows

   $$\begin{cases} G_\sigma(q) \wedge G_{\sigma'}(p) \Rightarrow (q,p) \to (q^+,p^+) & P(\sigma) = P(\sigma') \neq \varepsilon \\ G_\sigma(q) \Rightarrow (q,p) \to (q^+,p) & P(\sigma) = \varepsilon \\ G_{\sigma'}(p) \Rightarrow (q,p) \to (q,p^+) & P(\sigma') = \varepsilon \end{cases} \tag{4.5}$$

   Thus, when σ is observable, a synchronized transition is occurred and when it is unobservable an asynchronous transition occurs only in one of the automata.

3. Use a model-checking logic to check whether there exists an ambiguous cycle in the synchronized automata or not. In our case a CTL specification for diagnosability verification is proposed in (4.3) which here results in the specification.

   $$\neg[EF\, EG(q = q_0 \wedge p = p_0 \wedge B(q_0) \wedge \neg B(p_0))]. \tag{4.6}$$

   As it is explained in (4.3), the above specification is equal to

   $$[AG\, AF \neg(q = q_0 \wedge p = p_0 \wedge B(q_0) \wedge \neg B(p_0))].$$

   If the above formula holds for the synchronized automata, the model is diagnosable. Otherwise, it means that there exists a pair of event traces in the model, one containing a failure and the other one containing no failure.

   An LTL specification for diagnosability based on the above specification is formulated as

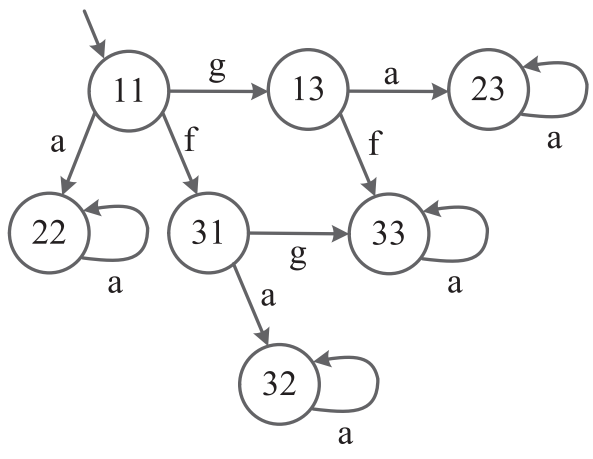
   $$\forall q_0,p_0[GF\neg(q = q_0 \wedge p = p_0 \wedge B(q_0) \wedge \neg B(p_0))],$$

   where we remind that LTL is based on the quantifier "for all".

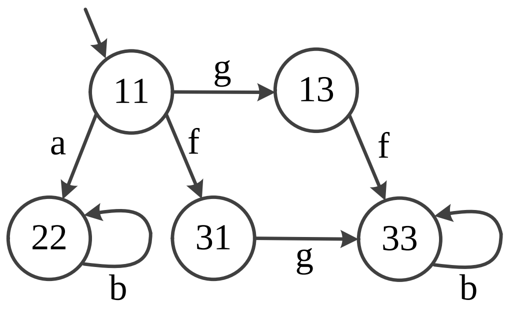
   In [5], specification in first order LTL is introduced in the following

   $$\exists q_0,p_0[EGF(q = q_0 \wedge p = p_0 \wedge B(q_0) \wedge \neg B(p_0))].$$

   Since this specification is not available in NuSMV, the CTL specification introduced in this thesis is implemented.



(a) The projected synchronous composition of the non-diagnosable automaton in Fig. 4.1(a)



(b) The projected synchronous composition of the diagnosable automaton in Fig. 4.1(b)

Figure 4.12: The projected synchronous composition of (a) the automaton in Fig. 4.1(a), (b) the automaton in Fig. 4.1(b).

Example 4.4

In this example the projected synchronous composition is illustrated for both the non-diagnosable and the diagnosable automata depicted in Fig. 4.1(a) and Fig. 4.1(b), respectively. For simplicity, the states of the synchronized automaton are coded by integers and for instance, state $\langle 1{,}3\rangle$ is depicted as 13. Moreover, events a and b are observable events and f and g are unobservable events. The unobservable event g belongs to the copy of the automaton corresponding to the event f, to enable us to illustrate the interleaving behavior of the projected synchronous composition for the unobservable events. Note that states 23 and 32 of the automaton in Fig. 4.12(a) have different values for $B(q)$. Thus, (4.6) does not hold for the two states which means that the automaton is not diagnosable, while (4.6) holds for the synchronized automaton in Fig. 4.12(b).

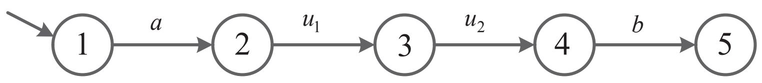


Figure 4.13: The considered automaton for the Example 4.3.

4.6 Online Diagnosis Algorithm

The online diagnosis algorithm which is introduced in [5] can be implemented, even in the case that the system is not diagnosable. In this case some failures are missed to report. There are some notations which are used in the algorithm formulation. For this purpose a predicate $N_k(q)$ is defined, which shows the possible next states following the occurrence of the kth observable event. fr implies the set of forward one-step reachable states, meaning the next states after occurrence of a transition. fr^* is the symbol of forward reachability, and denotes the set of states which are reachable from a specific state by executing zero or more transitions. Moreover, $P^{-1}(\varepsilon)$ is the inverse projection, with P as the defined projection, and indicates the unobservable events that are projected to ε. Similarly, $P^{-1}(\delta)$ indicates the observable events.

- Initial step:
$$N_0(q) = fr^*{}_{P^{-1}(\varepsilon)}\left[I(q)\right] \tag{4.7}$$
Starting from the initial state $I(q)$, $N_0(q)$ checks for the possible forward reachable states through strings of unobservable events before the next observable event. In other words, $N_0(q)$ consists of a set of states reached by zero or more unobservable events starting from $I(q)$.

- Iteration step:
$$N_{k+1}(q) = fr^*{}_{P^{-1}(\varepsilon)}\left[fr_{P^{-1}(\delta)}\left(N_k(q)\right)\right] \tag{4.8}$$
In this step, the inner part, $\left[fr_{P^{-1}(\delta)}\left(N_k(q)\right)\right]$, is the set of one-step forward reachable states, starting from the states in $N_k(q)$ and following all the single observable events ($P^{-1}(\delta) = \sigma$). Then starting from the inner part, fr^* is all the zero or more reachable states following the occurrence of unobservable events.

  In summary, in each iteration, the algorithm checks for the next single observable event and updates the state set upon occurrence of that event. Then, starting from the updated state set, it checks for zero or more unobservable events and updates the state set by executing the events. This algorithm iterates as long as it introduces a predicate that have not been visited before. An example is illustrated in the sequel.

Example 4.5

To clarify the algorithm, fr and fr^* are illustrated in the Fig. 4.13. Here I(q) = [1], $P^{-1}(\varepsilon) = \{u_1,u_2\}$ and $P^{-1}(\delta) = \{a,b\}$. As it is shown in (4.9), there is no transition with unobservable event starting from the initial state. Thus, I(q) and $N_0(q)$ are the same.

$$N_0(q) = fr^*{}_{P^{-1}(\varepsilon)}[I(q)] = fr^*_{\{u_1,u_2\}}[1] = [1] \tag{4.9}$$

In the iteration step, the inner part of (4.8) is

$$fr_{p^{-1}(\delta)}(N_k(q)) = fr_a(N_0(q)) = fr_a([1]) = [2] \tag{4.10}$$

and hence, $N_1(q)$ is

$$N_1(q) = fr^*{}_{P^{-1}(\varepsilon)}[2] = fr^*_{\{u_1,u_2\}}[2] = [2{,}3{,}4]. \tag{4.11}$$

As the next iteration step, to calculate the reachable states in $N_2(q)$, the inner part of (4.8) is

$$fr_{p^{-1}(\delta)}(N_k(q)) = fr_b(N_1(q)) = fr_b([2{,}3{,}4]) = [5]$$

and since there is no unobservable event initiating from state 5, $N_2(q)$ is

$$N_2(q) = fr^*{}_{P^{-1}(\varepsilon)}[5] = fr^*_{\{u_1,u_2\}}[5] = [5].$$

Note

In each iteration, the state set is checked regarding two aspects;

1. The possibility of occurrence of event strings according to the system language, i.e., the event strings should be a subset of the prefix closure of the system language. In each iteration the algorithm is performed for all observable events, regardless of their occurrence possibility. Then, removing the impossible event strings and their corresponding states from the state set, the next possible state set is achieved.

2. The faulty state predicate is considered $B(q)$. Based on this, three different interpretations "ambiguous", "no failure" and "a failure has occurred in the past", can be the result. For each achieved state set during the iteration, if the predicate

$$\text{false} \neq N_k(q) \to B(q)$$

holds, it means that a failure has occurred in the past. Moreover,

$$\text{false} \neq N_k(q) \to \neg B(q)$$

means that no failure has occurred so far and otherwise, it is ambiguous.

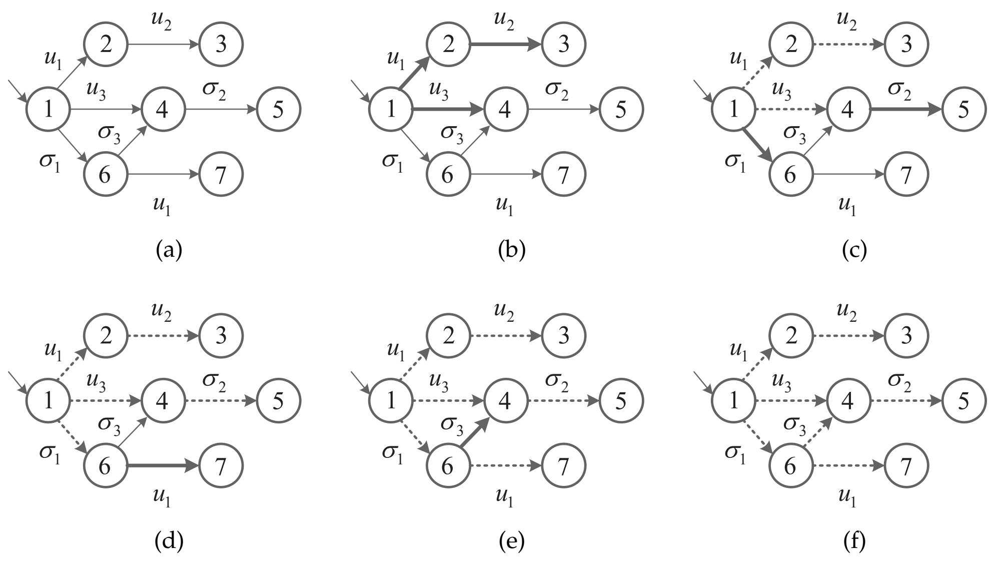


Figure 4.14: Online diagnosis.

Example 4.6

Equations (4.7) and (4.8) are illustrated in Fig. 4.14. σ_1, σ_2 and σ_3 are observable events, and u_1, u_2 and u_3 are unobservable events. Figure 4.14(a) shows the automaton where the algorithm of online diagnosis is performed. Starting from the initial state and following (4.7), states 1, 2, 3 and 4 are possible forward reachable states where the corresponding transitions are depicted in Fig. 4.14(b) by thick arrows. In Fig. 4.14(c), (4.8) is applied on the reached states from the previous step. In this step only single observable events are considered which are shown in thick arrows. The transitions belonging to the previous step are shown in dotted arrows. States 5 and 6 are reached in this step. In Fig. 4.14(d), starting from state 6, only state 7 is reachable by an unobservable event. Note that Fig. 4.14(c) depicts the inner part of (4.8), while Fig. 4.14(d) depicts the outer part. Furthermore, Figs. 4.14(c) and 4.14(d) together show one iteration step of the algorithm. Thus, in this step, states 6 and 7, and state 5 are reached upon occurrence of the observable events σ_1 and σ_2, respectively. Fig. 4.14(e) depicts the next iteration of the algorithm which starts based on the reached states in the previous step. Although in Fig. 4.14(f) all the states are reached, the algorithm continues iterating because the paths leading to the same states may differ. Therefore, the algorithm iterates as long as it introduces new states. Remember that states are augmented with labels, which may have different values based on the path towards them.

Example 4.7

Here is another example of implementing the online diagnosability method [5] on the automaton in the Fig. 4.15. In this automaton the observable event set is $\{a,b,c\}$,

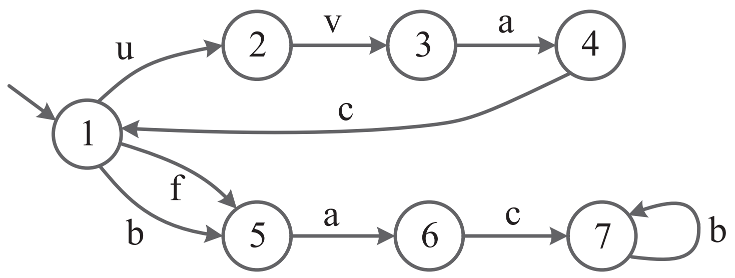


Figure 4.15: The considered automaton for the Example (4.7).

the unobservable event set is $\{f,u,v\}$ and f is a failure event. Note that although there is a cycle of uncertain states in the diagnoser, the uncertain states are associated only with the cycle 1, 2, 3, 4, 1 in the system automaton which all of them are labeled $F = 0$ in the cycle of uncertain states. Thus, the system is diagnosable. In (4.12), the possible initial states are shown with different values for F.

$$\begin{aligned} &k = 0 \\ &N_0(q) = [1,F = 0] \vee [2,F = 0] \vee [3,F = 0] \vee [5,F = 1] \end{aligned} \tag{4.12}$$

Based on the states obtained in (4.12), two observable events can be executed which are shown in (4.13). For the other observable event c, $N_1(q)$ predicate is false. Observing event b, implies that a failure has occurred because all labels are $F = 1$. However, observing event a, the diagnoser is still ambiguous.

$$\begin{aligned} &k = 1 \\ &\begin{cases} a \to N_1(q) = [4,F = 0] \vee [6,F = 1] \\ b \to N_1(q) = [1,F = 1] \vee [2,F = 1] \vee [3,F = 1] \vee [5,F = 1] \\ c \to N_1(q) = \text{False} \end{cases} \end{aligned} \tag{4.13}$$

Since in (4.13) two $N_1(q)$ predicates were obtained for a and b, all observable events for each of the two predicates should be checked. In (4.14), the $a \to c$ event trace is still ambiguous but both predicates obtained from the event traces $b \to a$ and $b \to b$ are certain that a failure has occurred previously. As it is seen, the $b \to b$ event trace has the same predicate as b event trace. Thus, we will not go further on this event trace in the following steps.

$$\begin{aligned} &k = 2 \\ &\begin{cases} a \to & \begin{cases} c \to N_2(q) = [1,F = 0] \vee [2,F = 0] \vee [3,F = 0] \vee [5,F = 1] \vee [7,F = 1] \\ a,b \to N_2(q) = \text{False} \end{cases} \\ b \to & \begin{cases} a \to N_2(q) = [4,F = 1] \vee [6,F = 1] \\ b \to N_2(q) = [1,F = 1] \vee [2,F = 1] \vee [3,F = 1] \vee [5,F = 1] = N_1(q) \\ c \to N_2(q) = \text{False} \end{cases} \end{cases} \end{aligned} \tag{4.14}$$

In (4.15), although the two new predicates imply that a failure has occurred, the iteration should be continued until reaching no new predicates.

$$k = 3$$
$$\begin{cases} a \rightarrow c \rightarrow \begin{cases} a \rightarrow N_3(q) = [4,F=0] \vee [6,F=1] = N_1(q) \\ b \rightarrow N_3(q) = [1,F=1] \vee [2,F=1] \vee [3,F=1] \vee [5,F=1] \vee [7,F=1] \\ c \rightarrow N_3(q) = \text{False} \end{cases} \\ b \rightarrow a \rightarrow \begin{cases} c \rightarrow N_3(q) = [1,F=1] \vee [2,F=1] \vee [3,F=1] \vee [7,F=1] \\ a,b \rightarrow N_3(q) = \text{False} \end{cases} \end{cases} \tag{4.15}$$

In (4.16), the algorithm stops checking the event trace $a \rightarrow c \rightarrow b$ because both predicates were seen before, then it remains traversing the trace $b \rightarrow a \rightarrow c$.

$$k = 4$$
$$\begin{cases} a \rightarrow c \rightarrow b \rightarrow \begin{cases} a \rightarrow N_4(q) = [4,F=1] \vee [6,F=1] = N_2(q) \\ b \rightarrow N_4(q) = [1,F=1] \vee [2,F=1] \vee [3,F=1] \vee [5,F=1] \\ \vee [7,F=1] = N_3(q) \\ c \rightarrow N_4(q) = \text{False} \end{cases} \\ b \rightarrow a \rightarrow c \rightarrow \begin{cases} a \rightarrow N_4(q) = [4,F=1] \vee [1,F=1] \\ b \rightarrow N_4(q) = [7,F=1] \\ c \rightarrow N_4(q) = \text{False} \end{cases} \end{cases} \tag{4.16}$$

The step (4.17) is the last step, and all the predicates imply that a failure has occurred some time in the past.

$$k = 5$$
$$\begin{cases} b \rightarrow a \rightarrow c \rightarrow a \rightarrow \begin{cases} c \rightarrow N_5(q) = [1,F=1] \vee [2,F=1] \vee [3,F=1] \vee [5,F=1] = N_1(q) \\ b \rightarrow N_5(q) = [7,F=1] = N_4(q) \\ a \rightarrow N_5(q) = \text{False} \end{cases} \\ b \rightarrow a \rightarrow c \rightarrow b \rightarrow \begin{cases} b \rightarrow N_5(q) = [7,F=1] = N_4(q) \\ a,c \rightarrow N_5(q) = \text{False} \end{cases} \end{cases} \tag{4.17}$$

4.7 Conclusion

In this chapter, diagnosability, failure diagnosis and diagnoser notions were explained. The concepts of uncertain and indeterminate cycles were described, which are the main notions in diagnosability test algorithms. Then two diagnosability tests along with a n online diagnosis algorithm were illustrated. In this perspective, temporal logic is explained and two CTL specifications for model checking were proposed. These concepts and algorithms are applied on the EFA and synchronization of EFAs in the next chapter, which are implemented in the NuSMV software described in chapter 6.

5

Extended Finite Automata

5.1 Introduction

To make the ordinary automaton more powerful in representing the system and make the model more compact Extended Finite Automaton (EFA) is introduced. EFAs enable adding different concepts to the model, for instance, time, time delay or logic conditions. An EFA is an ordinary automaton that is augmented by variables. In the definition of an EFA introduced in [8], guard expressions and action functions are defined as they are associated to the transitions. A guard predicate can enable its corresponding transition, if and only if it is evaluated to `true`. Then, the transition is executed and the variables of the action function are updated. Here, instead of guards and actions, condition relations are defined which include both guards and actions. The following section describes this in more details.

5.2 EFA

An extended finite-state automaton E is a 6-tuple

$$E = \langle X,\Sigma,C,T,I,M \rangle$$

where

(i) $x_i \in X_i$ is a state variable and $X = X_1 \times \ldots \times X_n$ is the domain of definition of an n-tuple of state variables; this domain can also be divided as $X = L \times V \times \mathcal{F}$, where L is the set of locations which are local variables, V is the set of global variables, and $\mathcal{F}$ is the set of failure states,

(ii) $\Sigma = \{\sigma_1, \ldots ,\sigma_m\}$ is a non-empty set of events,

(iii) $C = \{C_{\sigma_1}, \ldots ,C_{\sigma_m}\} : X \times X \to \mathbb{B}$ is the set of condition predicates, including both the variables and their updated values,

(iv) $T = \{T_{\sigma_1}, \ldots, T_{\sigma_m}\} : X \times \Sigma \times C \times X \to \mathbb{B}$ is the transition predicate,

(v) $I : X \to \mathbb{B}$ is the predicates on initial states, and

(vi) $M : X \to \mathbb{B}$ is the predicates on marked states.

In the following the 6-tuple E is presented in more details.

Example 5.1

For $n = 2$, the predicate of initial states can be expressed e.g. as

$$I : x_1 = 0 \wedge x_2 = 2,$$

and the predicate of marked states as

$$M : x_1 = 1,$$

which means that $x_1 = 1$ and all possible values of x_2 are accepted as marked states.

Condition Predicates

A condition $C_\sigma(x,x^+)$ is a predicate over the variables which includes the variable and its updated value. For nondeterministic EFAs, with n_σ transitions over event σ, the condition is the disjunction of all conditions over the transitions as shown in (5.1). The transitions may start from one state or different states, where $C_{\sigma,j}(x,x^+) : X \times X \to \mathbb{B}$ and $j = 1, \ldots, n_\sigma$.

$$C_\sigma(x,x^+) = \bigvee_{j=1}^{n_\sigma} C_{\sigma,j}(x,x^+). \tag{5.1}$$

Index Set

The index set is defined as a set containing the indices of the variables which are not updated by the implicit actions in the conditions.

Introduce the notion

$$x^+_{\xi,i} = \langle x^+_1, \ldots, x^+_{i-1}, \xi, x^+_{i+1}, \ldots, x^+_n \rangle,$$

then the index set is

$$\Omega(C_{\sigma,j}) = \{i | \exists x^+ [C_{\sigma,j}(\perp, x^+_{\xi,i})]\} \tag{5.2}$$

where $i \in \{1, \ldots, n\}$.

The symbol ξ represents implicit action in the condition that do not update the value of variable. When $C_{\sigma,j}$ involves a condition on x^+_i then $C_{\sigma,j}(x,x^+_{\xi_i}) = C_{\sigma,j}(x,x^+)$, while $C_{\sigma,j}(x,x^+_{\xi_i}) = C_{\sigma,j}(x,x^+)$ when there is no condition on x^+_i. In other words, if $x^+_i = \xi$, it means that the $C_{\sigma,j}(x,x^+_{\xi_i}) = C_{\sigma,j}(x,x^+)$ and x^+_i is not updated.

If there is no condition on x^+ in $C_{\sigma,j}(x,x^+)$, it is considered as $x^+ = \Xi$, which implies that in the specific transition no variable is updated.

Example 5.2

The following index sets are presented to illustrate (5.2) intuitively. Then

$$\begin{aligned}
&\Omega(x_2 = 0 \wedge x_1^+ = 1) = \{2\},\\
&\Omega(x_1 = 1 \wedge x_2 = 0) = \{1,2\},\\
&\Omega(x_1^+ = 0 \wedge x_2^+ = 1) = \emptyset.
\end{aligned}$$

Example 5.4

Consider the following condition relation which is depicted in Fig. 5.1(a).

$$C_\sigma(x,x^+) := l = 0 \wedge l^+ = 1 \wedge (v_1^+ = 1 \vee v_1^+ = 2 \vee v_1^+ = 3 \vee v_2^+ = 1)$$

where $X = I_3 \times I_1$ for $I_n = \{0,1,\dots,n\}$. The above condition can be handled as two expressions

$$\begin{aligned}
C_{\sigma,1} &:= (1 \leqslant v_1^+ \leqslant 3) \wedge l = 0 \wedge l^+ = 1\\
C_{\sigma,2} &:= v_2^+ = 1 \wedge l = 0 \wedge l^+ = 1.
\end{aligned}$$

Therefore, as shown in Fig. 5.1(b), it can be rewritten as

$$C_\sigma(x,x^+) = C_{\sigma,1} \vee C_{\sigma,2}.$$

It gives again

$$\begin{aligned}
\Omega(C_{\sigma,1}(x,x^+)) &= \{2\}\\
\Omega(C_{\sigma,2}(x,x^+)) &= \{1\}
\end{aligned}$$

for all x and x^+ as illustrated in Fig. 5.1(c).

Transition Predicates

Transition predicate is

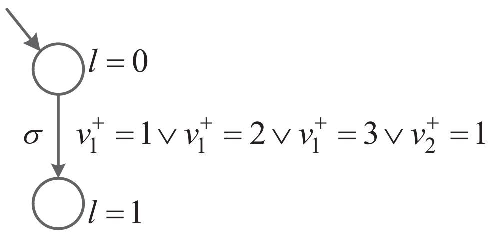
$$T(x,x^+) = \bigvee_{j=1}^{n_\sigma} T_{\sigma,j}(x,x^+) \tag{5.3}$$

where $T_\sigma(x,x^+) : X \times X \to \mathbb{B}$ and $T_{\sigma,j}(x,x^+)$ is

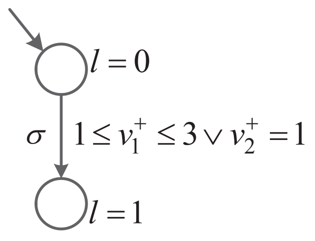
$$T_{\sigma,j}(x,x^+) = C_{\sigma,j}(x,x^+) \bigwedge_{i \in \Omega(C_{\sigma,j})} x_i^+ = x_i \wedge e = \sigma \tag{5.4}$$

The transition predicate in (5.3) is a set of disjunction of conjunctive clauses which contains the current values of the implicit actions, along with all conditions that are found on a specific event. In (5.4), $C_j(x,x^+)$ must be true for some (x,x^+), since otherwise the transition can not be executed.

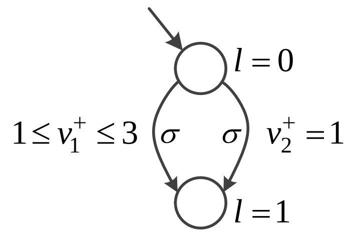
The difference between the transition predicate in (5.3) and the condition in (5.1) is the conjunctive part in (5.4), which keeps the current value of x_i as the updating value for x_i^+ for those i that belongs to the index set.



(a) $C_\sigma(x,x^+)$



(b) $C_\sigma(x,x^+)$



(c) $\tilde{C}_{\sigma,1} \vee \tilde{C}_{\sigma,2}$

Figure 5.1: (a), (b) Two different representations of the condition predicate $C_\sigma(x,x^+) := v_1^+ = 1 \vee v_1^+ = 2 \vee v_1^+ = 3 \vee v_2^+ = 1$, (c) Two separated condition predicates $\tilde{C}_{\sigma,1} := 1 \le v_1^+ \le 3$ or $\tilde{C}_{\sigma,2} := v_2^+ = 1$.

Nondeterministic EFA

In deterministic EFAs, at most one transition is enabled at any time. On the contrary, nondeterministic EFAs allow multiple possible transitions. This notion is explained in the following example. In other words, if $\exists x, C_{\sigma,j_1}(x,x^+)$ for some $\sigma \in \Sigma$, and more than one value of x^+ is possible then that transition is nondeterministic.

Example 5.5

Consider Fig. 5.2 as an EFA, the σ-transition from location $l = 1$ to $l = 2$ is possible for the variable $v \in \{0,1,2,3\}$, and the σ-transition from location $l = 1$ to $l = 3$ is possible for the variable $v \in \{2,3,4\}$. If v is in the common range, i.e., $v \in \{2,3\}$, the EFA is nondeterministic, because from the initial state $l = 1$, two transitions can be executed and the next state is ambiguous and would be nondeterministically chosen between one of the locations $l = 2$ or $l = 3$. On the other hand, if $v \in \{0,1,4\}$ only one of the σ-transitions is enabled, and in this case, the EFA is deterministic.

5.3 Extended Full Synchronous Composition

Let $E^1 \parallel E^2 = \langle X, \Sigma^1 \cup \Sigma^2, C, T, I^1 \wedge I^2, M^1 \wedge M^2 \rangle$ be the Extended Full Synchronous Composition (EFSC) of two EFAs, where

(i) X is the extended finite set of states containing both global and local variables.

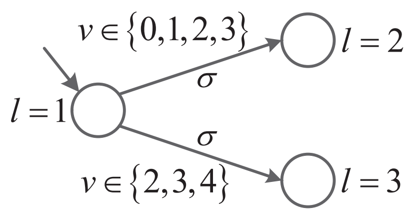


Figure 5.2: A nondeterministic EFA.

(ii) $\Sigma = \Sigma^1 \cup \Sigma^2$ is the union of all events of the two EFAs.

(iii) The elements in C are defined as

$$C_{\sigma,j} = \begin{cases} C^1_{\sigma,j_1} \wedge C^2_{\sigma,j_2} & \sigma \in \Sigma^1 \cap \Sigma^2 \\ C^1_{\sigma,j_1} & \sigma \in \Sigma^1 \backslash \Sigma^2 \\ C^2_{\sigma,j_2} & \sigma \in \Sigma^1 \backslash \Sigma^2 \end{cases} \tag{5.5}$$

where $C^1_{\sigma,j_1} \in C^1$, $C^2_{\sigma,j_2} \in C^2$, and $j_1 = 1,\ldots,n^1_\sigma$ and $j_2 = 1,\ldots,n^2_\sigma$.

(iv) T is the set of transition predicates defined according to (5.3) and (5.4).

(v) $I(x) = I^1(x) \wedge I^2(x)$ is the conjunctive set of predicates on initial states.

(vi) $M(x) = M^1(x) \wedge M^2(x)$ is the conjunctive set of predicates on marked states.

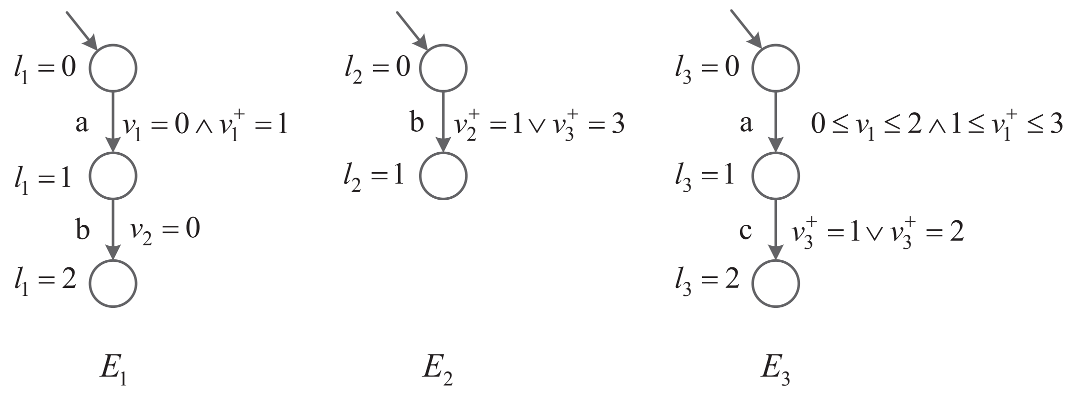
The transition relation is the combination of all conditions, attached to corresponding transitions of event σ, along with the variables for which there is no updating condition. Observe that the transition relation depends on C and not T^k. When for some i in x_i^+ there is no condition on x_i^+, they are not updated and keep their current value.

In the conflicting case, where the conditions on a variable explicitly try to update it to different values, the variable is not updated. In other words, no transition is executed and it is another interpretation of "keep current value". The following example shows it clearly.

Example 5.6

There are three EFAs in Fig. 5.3 for which the transition relation of their synchronization $E_1 \parallel E_2 \parallel E_3$ is written below. Domains of the variables are $v_1 \in \{0,1,2,3\}$, $v_2 \in \{0,1\}$ and $v_3 \in \{0,1,2,3\}$. Following equation (5.3), $T_\sigma(x,x^+)$ for each event is

$$T_a : (l_1 = 0 \wedge l_3 = 0 \wedge l_1^+ = 1 \wedge l_2^+ = l_2 \wedge l_3^+ = 1) \wedge$$
$$(v_1 = 0 \wedge v_1^+ = 1 \wedge v_2^+ = v_2 \wedge v_3^+ = v_3),$$



$l_1 = 0$
a
$v_1 = 0 \wedge v_1^+ = 1$
$l_1 = 1$
b
$v_2 = 0$
$l_1 = 2$
E_1
$l_2 = 0$
b
$v_2^+ = 1 \vee v_3^+ = 3$
$l_2 = 1$
E_2
$l_3 = 0$
a
$0 \le v_1 \le 2 \wedge 1 \le v_1^+ \le 3$
$l_3 = 1$
c
$v_3^+ = 1 \vee v_3^+ = 2$
$l_3 = 2$
E_3

Figure 5.3: Three EFAs considered for synchronization.

$$\begin{gathered}T_b : ((l_1 = 1 \wedge l_2 = 0 \wedge l_1^+ = 2 \wedge l_2^+ = 1 \wedge l_3^+ = l_3)\vee \\ ((v_2 = 0 \wedge v_1^+ = v_1 \wedge v_2^+ = 1 \wedge v_3^+ = v_3)\vee \\ (v_2 = 0 \wedge v_1^+ = v_1 \wedge v_2^+ = v_2 \wedge v_3^+ = 1)),\end{gathered}$$

$$\begin{gathered}T_c : (l_3 = 1 \wedge l_1^+ = l_1 \wedge l_2^+ = l_2 \wedge l_3^+ = 2)\vee \\ (v_1^+ = v_1 \wedge v_2^+ = v_2 \wedge (v_3^+ = 1 \vee v_3^+ = 2)).\end{gathered}$$

As an example, based on the EFAs in Fig. 5.3, the transitions on event $a \in \Sigma^1 \cap \Sigma^3$ can be executed merely when both variables are updating to the same value as in $C_a^1 \wedge C_a^3$, i.e., $v_1^+ = 1$. Otherwise, there is a conflict and the synchronization stops in its initial state.

5.4 Conclusion

In this chapter automata extended with variables are introduced as extended finite automata (EFAs). Different parts of its 6-tuple were explained through expressive examples. Nondeterministic EFA was also illustrated through examples where different conjunctive or disjunctive predicates in conditions augmented to transitions were shown. Moreover, extended full synchronous composition in EFAs was explained by an example.

6

THE NUSMV MODEL CHECKER

6.1 Introduction

Verifying the satisfaction of a CTL formula ϕ, by a system A, is known as *model-checking*, while verifying the existence of a system A such that it satisfies a given formula is known as *satisfiability* problem. NuSMV is a model-checker software tool that verifies the formula provided each state variable has a bounded domain and is originated from the extension and reimplementation of SMV.

NuSMV has different capabilities to represent synchronous and asynchronous Finite State Machines (FSMs), and also to analyze specifications expressed by CTL or LTL, using SAT-based and Binary Decision Diagram-based (BDD-based) model checking techniques. The specification represents the behavior of the FSM as indicated by "possible next state" relations. The possible next states and transitions are determined using the values of variables and the updates of the variables, respectively. A specification is represented by the main sections; `VAR`, `ASSIGN` and `CTL` (or `LTL`) expressions [13], [14].

In this chapter a diagnosability test algorithm is implemented on both ordinary automaton and EFAs. For the former, the algorithm is presented in one main module and also with sub-modules. For the later, synchronization of three nondeterministic EFAs is presented and then the diagnosability algorithm is applied on the synchronization of NEFAs as another example. The code can be implemented on an arbitrary number of EFAs. Moreover, two algorithms for the two later implementations are presented.

6.2 Diagnosability Test on Ordinary Automaton

The diagnosability test algorithm 2 described in the Section 4.5 is implemented on the automaton in the Fig. 4.1(b). The considered CTL specification is based on the temporal logic definition for failure diagnosis, which is described in the Section 4.4. As it is expected for a diagnosable automaton, the `SPEC EF EG (F1!=F2)` and `SPEC AG AF (F1=F2)` are `false` and `true`, respectively. Two NuSMV codes are presented for this algorithm which are implemented on the same automaton depicted in Fig. 4.1(b).

Diagnosability Test on Ordinary Automaton with One Main Module

The first code only contains a main module where the algorithm is implemented sequentially, i.e., variable and transition definition of the automaton and its copy, synchronization of them, specifying the next values to variables and verifying the specification.

In the code, all the variables are defined in the `VAR` section. `IVAR` shows the input variables, i.e. the events. In `DEFINE` part, each transition of each automaton is named. Remember that in the diagnosability test algorithm 2, the automaton should be projected synchronized with a copy of itself. Therefore, two similar automata with different variables are defined here. `TRANS` performs the synchronization, followed by the new value assignments, and in the end, `SPEC` comes which verifies the specification.

NuSMV Code

```
MODULE main

VAR
v :  1..3;
w :  1..3;
F1 :  0..1;
F2 :  0..1;

IVAR
event :  {a,b,f,g};

INIT
v=1 & w=1 & F1=0 & F2=0;

DEFINE
tv1 := v=1 & (event=a);
tv2 := v=1 & (event=f);
tv3 := v=2 & (event=b);
tv4 := v=3 & (event=b);
tv5 := (event=g);

tw1 := w=1 & (event=a);
tw2 := w=1 & (event=g);
tw3 := w=2 & (event=b);
tw4 := w=3 & (event=b);
tw5 := (event=f);
```

```
TRANS
((tv1) xor (tv2) xor (tv3) xor (tv4) xor (tv5)) &
((tw1) xor (tw2) xor (tw3) xor (tw4) xor (tw5)) &
((tv1 xor tv3) & (tw1 xor tw3)) xor ((tv4 & tw4)) xor ((tv2 xor tw2));

DEFINE
next_v := case
 tv1 | tv3 :  2;
 tv2 | tv4 :  3;
 TRUE : v;
esac;

next_w := case
 tw1 | tw3 :  2;
 tw2 | tw4 :  3;
 TRUE : w;
esac;

next_F1 := case
 tv2 :  1;
 TRUE : F1;
esac;

next_F2 := case
 tw2 :  1;
 TRUE : F2;
esac;

TRANS
next(v) = next_v & next(w) = next_w &
next(F1) = next_F1 & next(F2) = next_F2;

SPEC
AG AF (F1=F2)
```

Diagnosability Test on Ordinary Automaton with Two Modules

The second code, contains two modules, where the automaton depicted in Fig. 4.1(b) is defined in the module `automaton`, and the projected synchronization is defined in the module `main`. The projected synchronization is performed by the help of `int_leav` input variable, which stands for the "interleaving" behavior. It chooses the value 1 or 2, arbitrarily, and implies either the transition with unobservable event of the first automaton should be executed or the transition with unobservable event of the second one. This means that the unobservable events are not executed synchronously. `SigmaO` and `SigmaU` represent the observable and unobservable events, respectively.

NuSMV Code

```
MODULE automaton(q,Fa,event,int_leav,k12)

DEFINE
SigmaO:={a,b};
SigmaU:={f};
t1 := q=1 & (event=a);
t2 := q=1 & (event=f) & int_leav=k12;
t3 := q=2 & (event=b);
t4 := q=3 & (event=b);
t0 := !(event in SigmaO | event in SigmaU & int_leav=k12);

next_l:=case
  t1 | t3 :  2;
  t2 | t4 :  3;
  t0 :  q;
  TRUE : -1;
esac;

next_F := case
  t2 :  1;
  TRUE : Fa;
esac;
---------------------------
MODULE main

VAR
F1 :  0..1;
F2 :  0..1;
l1 :  1..3;
l2 :  1..3;
G1 :  automaton(l1,F1,event,int_leav,1);
G2 :  automaton(l2,F2,event,int_leav,2);

IVAR
event :  {a,b,f};
int_leav :  {1,2};

INIT
l1=1 & l2=1 & F1=0 & F2=0 ;

TRANS
next(l1)=G1.next_l & G1.next_l!=-1 &
next(l2)=G2.next_l & G2.next_l!=-1 &
next(F1)=G1.next_F & next(F2)=G2.next_F;
```

```
SPEC
AG AF (F1=F2)
```

6.3 Synchronization of EFAs

This section is on the synchronization of nondeterministic EFAs where an arbitrary number of EFAs can be synchronized. As an example, the EFAs in Fig. 5.3 are synchronized which are defined in `EFA1`, `EFA2` and `EFA3` modules. To be more precise, module `EFA3` will be explained, which represents EFA E_3 in Fig. 5.3 with a slight change in the guard condition of the second transition where a condition on location of E_1 is added. Figure 6.1, shows the EFA representation of module EFA3. Note that $1 \leq v_1^+ \leq 3$ in the a-transition, is the compact form for representing three transitions, where their difference is in the next possible value of v_1, which nondeterministically can be assigned. Moreover, in the three transitions $v_2^+ = \xi$ and $v_3^+ = \xi$. It is also the same for the c-transition. To be able to distinguish which transition is executed and what value should be assigned to the variable, an indicator m is defined in the code for such nondeterministic transitions.

After defining the transitions, the next location and also the next value of variables are assigned. In the `DEFINE` conditions, `TRUE` means "otherwise", which holds when all other conditions are false. Here ξ, which means the variable "does not care" about its next value, is represented by -1. Note that, the -1 in the `TRUE` condition of the location definition (`next_l`) is used only to make the case condition exhaustive and here it does not convey the meaning of ξ.

Note that actual parameters of a module can potentially be instances of other modules. Therefore, parameters of modules allow access to the components of other module instances, as in the example, an instance of the module `synch` is passed to the sub-module `synchnv`. In the module `main`, e.g., s_2 which is the synchronization of `EFA1` and `EFA2`, and is the input to the next `synch` function to be synchronized with `EFA3`, is declared to be an instance of the module `synch`. `synch` declares three instances of the module `synchnv`. Every instance of the `synchnv` module has a defined `nv` which specifies the conditions for updating to the next synchronous value. Thus, a `synchnv` needs access to the parent `synch` to access all the `synchnv` in the `synch`.

In the module `synchnv`, it checks different conditions that may happen during synchronization. Starting from the first condition, `keepv` checks whether a variable is equal to ξ in all automata or not. If it is true, the current value should be kept. Note that this condition is the only one that is checked globally in the `main` module. It considers all automata at the same time, while other conditions are considered locally in the `synchnv` module.

Consider automata E_1 and E_3 which have event a in their alphabet. In E_1 when event a is executed it updates the value of v_1 to 1, while in E_3 when event a is executed it updates the value of v_1 nondeterministically to 1, 2 or 3. By definition, in synchronization, when event a is executed the variable will be updated nondeterministically to the conjunction of the values indicated in each transition. This

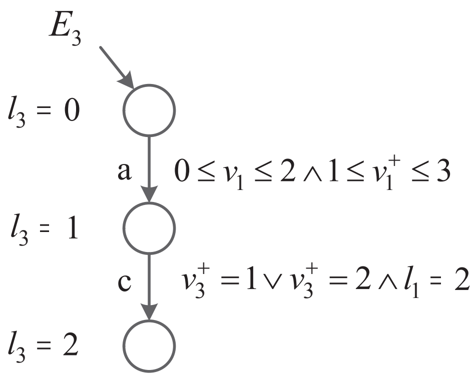

E_3
$l_3 = 0$
a
$0 \le v_1 \le 2 \wedge 1 \le v_1^+ \le 3$
$l_3 = 1$
c
$v_3^+ = 1 \vee v_3^+ = 2 \wedge l_1 = 2$
$l_3 = 2$

Figure 6.1: Automaton representation of the module EFA3.

definition is shown in the condition

$$nvE1! = -1 \ \& \ nvE2! = -1 \ \& \ nvE1 = nvE2 \quad : \quad nvE1;$$

which indicates that if the actions in both EFAs are not equal to ξ, and are equal to each other, one of them will be kept as the next value for the variable. Otherwise if neither of them are equal to ξ and both try to update the variable to different values, a conflict happens. Thus, the condition mentioned above explicitly performs conjunction on the updating conditions.

NuSMV Code

```
MODULE EFA1(l,v1,v2,v3,e)

DEFINE
t1:= l=0 & e=a & v1=0;
t2:= l=1 & e=b & v2=0;
t0:= !(e in {a,b});

next_l:= case
 t1 :  1;
 t2 :  2;
 t0 :  l;
 TRUE : -1;
esac;

next_v1.nv := case
 t1 :  1;
 TRUE : -1;
esac;
next_v2.nv := -1;
next_v3.nv := -1;
---------------------------
```

```
MODULE EFA2(l,v1,v2,v3,e)

IVAR
m :  {0,1};

DEFINE
t10:= l=0 & e=b & m=0;
t11:= l=0 & e=b & m=1;
t0:= !(e in {b});

next_l:= case
 t10 xor t11 :  1;
 t0 :  l;
 TRUE : -1;
esac;

next_v1.nv := -1;

next_v2.nv := case
 t10 :  1;
 TRUE : -1;
esac;

next_v3.nv := case
 t11 :  3;
 TRUE : -1;
esac;
---------------------------
MODULE EFA3(l,l1,v1,v2,v3,e)

IVAR
m :  {0,1,2};

DEFINE
t10:= l=0 & e=a & (v1=0 | v1=1 | v1=2) & m=0;
t11:= l=0 & e=a & (v1=0 | v1=1 | v1=2) & m=1;
t12:= l=0 & e=a & (v1=0 | v1=1 | v1=2) & m=2;
t20:= l=1 & e=c & l1=2 & m=0;
t21:= l=1 & e=c & l1=2 & m=1;
t0:= !(e in {a,c});

next_l:= case
 t10 | t11 | t12 :  1;
 t20 | t21 :  2;
 t0 :  l;
 TRUE : -1;
esac;
```

```
next_v1.nv := case
 t10 :  1;
 t11 :  2;
 t12 :  3;
 TRUE : -1;
esac;
next_v2.nv := -1;
next_v3.nv := case
 t20 :  1;
 t21 :  2;
 TRUE : -1;
esac;
---------------------------
MODULE synchnv(nvE1,nvE2,keepv,v)

DEFINE
nv := case
 keepv :  v;
 nvE1!=-1 & nvE2!= -1 & nvE1 = nvE2 :  nvE1;
 nvE1 =-1 & nvE2!= -1 :  nvE2;
 nvE1!=-1 & nvE2 = -1 :  nvE1;
 TRUE : -1;
esac;
---------------------------
MODULE synch(E1,E2,keepv1,keepv2,keepv3,v1,v2,v3)

VAR
next_v1 :  synchnv(E1.next_v1.nv,E2.next_v1.nv,keepv1,v1);
next_v2 :  synchnv(E1.next_v2.nv,E2.next_v2.nv,keepv2,v2);
next_v3 :  synchnv(E1.next_v3.nv,E2.next_v3.nv,keepv3,v3);
---------------------------
MODULE main

VAR
v1 :  0..3;
v2 :  0..1;
v3 :  0..3;
l1 :  {0,1,2};
l2 :  {0,1};
l3 :  {0,1,2};
E1 :  EFA1 (l1,v1,v2,v3,e);
E2 :  EFA2 (l2,v1,v2,v3,e);
E3 :  EFA3 (l3,l1,v1,v2,v3,e);
s2 :  synch (E1,E2,keep_v1,keep_v2,keep_v3,v1,v2,v3);
```

```
s3 :  synch (s2,E3,keep_v1,keep_v2,keep_v3,v1,v2,v3);

IVAR
e :  {a,b,c};

INIT
v1=0 & v2=0 & v3=0 & l1=0 & l2=0 & l3=0;

DEFINE
keep_v1 := E1.next_v1.nv=-1 & E2.next_v1.nv=-1 & E3.next_v1.nv=-1;
keep_v2 := E1.next_v2.nv=-1 & E2.next_v2.nv=-1 & E3.next_v2.nv=-1;
keep_v3 := E1.next_v3.nv=-1 & E2.next_v3.nv=-1 & E3.next_v3.nv=-1;

TRANS
next(l1)=E1.next_l & next(l2)=E2.next_l & next(l3)=E3.next_l &
E1.next_l!=-1 & E2.next_l!=-1 & E3.next_l!=-1 &
next(v1)=s3.next_v1.nv & next(v2)=s3.next_v2.nv & next(v3)=s3.next_v3.nv;
```

Algorithm 1_ Synchronization of EFAs

I. **MODULE** $EFA^i = (l^i, v^1, \dots, v^J, e)$

(For $i = 1, \dots, I$, where I and J are number of EFAs and variables, respectively.

1. Define locally partial transition relations, including source location, event and guard conditions. (If the transition is nondeterministic, add indicator m for each distinct transition.)

$$t := (l = \text{source location}) \,\&\, (e = \text{event}) \,\&\, (\text{guard predicate});$$

2. $\forall\, v^j, l$ in each EFA define locally target locations ($next\_l$), and variables updated values ($next\_v^j$) in case condition assignments.

II. **MODULE synchnv (**$E^1.next\_v, E^2.next\_v, keep\_v, v$**)**

1. Define synchronization based on its definition for EFAs in Section (5.3) .

III. **MODULE synch (**$E^1, E^2, keep\_v^j, v^j$**)**

1. Define globally a next value for each variable, based on their current values in the two synchronizing EFAs. $\forall\, j$,

$$next\_v^j = synchnv(E^1.next\_v^j, E^2.next\_v^j, keep\_v^j, v^j)$$

IV. **MODULE main**

1. Define all l^i, v^j, E^i, S^2, and S^{i+1}, as well as event set and, initial values.

$$\begin{aligned} E^i &= EFA^i(l^i, v^1, \dots, v^J, e) \\ S^2 &= synch(E^1, E^2, v^1, \dots, v^J, keep\_v^1, \dots, keep\_v^J) \\ S^{i+1} &= synch(S^i, E^{i+1}, v^1, \dots, v^J, keep\_v^1, \dots, keep\_v^J) \end{aligned} \tag{6.1}$$

2. $\forall\, j$, define

$$keep\_v^j := \bigwedge_{i=1}^{I} (E^i.next\_v^j = -1)$$

3. $\forall\, v^j, l^i$, define globally total combined transition relations.

$$\bigwedge_{i=1}^{I} (next(l^i) = E^i.next\_l) \& \bigwedge_{j=1}^{J} (next(v^j) = S^I.next\_v^j)$$

Note that the guard conditions in each automaton can be on both global variables and local locations of other automata. In this case, in (6.1) the required local variables of the other automata should also be entered in E^i.

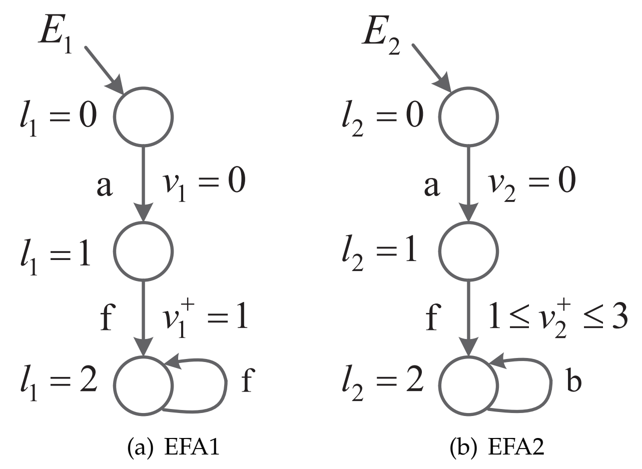


Figure 6.2: The two automata considered for synchronization.

6.4 Diagnosability Test on Synchronization of EFAs

In this section, two EFAs are synchronized, and the diagnosability test is performed on the synchronization. The two EFAs are depicted in Fig. 6.2. This code is for synchronization of two EFAs with two global variables, but it can be extended to a number of EFAs with more variables. Here, the same as the second diagnosability test code on ordinary automata, `int_leav` variable is defined to indicate the interleaving behavior in synchronization of transitions with unobservable events.

Module `main` includes all global variables, locations and failure variables, as well as two instances of the synchronization of EFAs and its copy, which are obtained through `EFAtot`. `EFAtot` declares instances of the modules `synch` and also `EFA1` and `EFA2`. Module `synch` declares two instances of module `synchnv`, because in this example there are two variables. Every instance of the `synchnv` module has a defined `nv`, which specifies the conditions for updating to the next synchronous value. A `synchnv` needs access to the parent `synch` to access all `synchnv` in the `synch`. Therefore, module `main` is a parent for module `EFAtot`, module `EFAtot` is a parent for `EFA1`, `EFA2` and `synch`, and module `synch` is a parent for `synchnv`.

NuSMV Code

```
MODULE EFA1(l,v1,v2,Fa,e,int_leav,k12)

DEFINE
SigmaO := {a};
SigmaU := {f};
t1 := l=0 & e=a & v1=0;
t2 := l=1 & e=f & int_leav = k12;
t3 := l=2 & e=f & int_leav = k12;
```

```
t0 := !(e in SigmaO | e in SigmaU & int_leav=k12);

next_l := case
 t1 :  1;
 t2 :  2;
 t3 :  2;
 t0 :  l;
 TRUE : -1;
esac;

next_v1.nv := case
 t2 :  1;
 TRUE : -1;
esac;

next_v2.nv := -1;

next_F := case
 t2 :  1;
 TRUE : Fa;
esac;
---------------------------
MODULE EFA2(l,v1,v2,Fa,e,int_leav,k12)

IVAR m :  {0,1,2};

DEFINE
SigmaO := {a,b};
SigmaU := {f};
t1 := l=0 & e=a & v2=0;
t20 := l=1 & e=f & int_leav=k12 & m=0;
t21 := l=1 & e=f & int_leav=k12 & m=1;
t22 := l=1 & e=f & int_leav=k12 & m=2;
t3 := l=2 & e=b;
t0 := !(e in SigmaO | e in SigmaU & int_leav=k12);

next_l := case
 t1 :  1;
 t20 | t21 | t22 :  2;
 t3 :  2;
 t0 :  l;
 TRUE : -1;
esac;

next_v1.nv := -1;

next_v2.nv := case
 t20 :  1;
 t21 :  2;
 t22 :  3;
 TRUE : -1;
esac;

next_F := case
 t20 | t21 | t22 :  1;
 TRUE : Fa;
esac;
---------------------------
MODULE synchnv(nvE1,nvE2,v,keepv)

DEFINE
nv := case
 keepv :  v;
 nvE1!=-1 & nvE2!=-1 & nvE1=nvE2 :  nvE1;
 nvE1 =-1 & nvE2!=-1 :  nvE2;
 nvE1!=-1 & nvE2 =-1 :  nvE1;
 TRUE : -1;
esac;
---------------------------
MODULE synch(E1,E2,v1,keepv1,v2,keepv2)

VAR
next_v1 :  synchnv(E1.next_v1.nv,E2.next_v1.nv,v1,keepv1);
next_v2 :  synchnv(E1.next_v2.nv,E2.next_v2.nv,v2,keepv2);
---------------------------
MODULE EFAtot(l1,l2,v1,v2,Fa,e,int_leav,k12)

VAR
E1 :  EFA1(l1,v1,v2,Fa,e,int_leav,k12);
E2 :  EFA2(l2,v1,v2,Fa,e,int_leav,k12);
SE : synch(E1,E2,v1,keep_v1,v2,keep_v2);

DEFINE
keep_v1 := E1.next_v1.nv=-1 & E2.next_v1.nv=-1;
keep_v2 := E1.next_v2.nv=-1 & E2.next_v2.nv=-1;

Faa:= case
 E1.next_F=0 & E2.next_F=0 :  0;
 TRUE : 1;
esac;

TRANS
next(l1)=E1.next_l & E1.next_l!=-1 &
next(l2)=E2.next_l & E2.next_l!=-1 &
next(v1)=SE.next_v1.nv & next(v2)=SE.next_v2.nv & next(Fa)= Faa;
```

```
---------------------------
MODULE main

VAR
l11 :  0..2;
l12 :  0..2;
v11 :  0..1;
v12 :  0..3;
F1 :  0..1;
l21 :  0..2;
l22 :  0..2;
v21 :  0..1;
v22 :  0..3;
F2 :  0..1;
Etot1 :  EFAtot(l11,l12,v11,v12,F1,e,int_leav,1);
Etot2 :  EFAtot(l21,l22,v21,v22,F2,e,int_leav,2);

IVAR
e :  {a,b,f};
int_leav :  {1,2};

INIT
l11=0 & l12=0 & v11=0 & v12=0 & F1=0 &
l21=0 & l22=0 & v21=0 & v22=0 & F2=0;

SPEC AG AF (F1 = F2)
```

Algorithm 2_ Diagnosability Test Algorithm on the Synchronization of EFAs

I. **MODULE** $EFA^i = (l^i,v^1,\ldots,v^J,Fa,e,int\_leav,k12)$

(For $i = 1,\ldots,I$, where I and J are number of EFAs and variables, respectively.

II. **MODULE synchnv (**$E^1.next\_v, E^2.next\_v, keep\_v, v$**)**

1. Define synchronization based on its definition for EFAs.

III. **MODULE synch (**$E^1, E^2, keep\_v^j, v^j$**)**

1. Define globally a next value for each variable, based on their current values in the two synchronizing EFAs. $\forall\, j$,

$$next\_v^j = synchnv(E^1.next\_v^j, E^2.next\_v^j, keep\_v^j, v^j)$$

IV. **MODULE EFAtot(**$l^1,\ldots,l^I,v^1,\ldots,v^J,Fa,e,int\_leav,k12)$

1. Define variables to declare each EFA sub-module and their synchronization. $\forall i, j$,

$$E^i = EFA^i(l^i,v^1,\ldots,v^J,Fa,e,int\_leav,k12)$$
$$S^2 = synch(E^1,E^2,v^1,\ldots,v^J,keep\_v^1,\ldots,keep\_v^J)$$
$$S^{i+1} = synch(S^i,E^{i+1},v^1,\ldots,v^J,keep\_v^1,\ldots,keep\_v^J)$$

2. $\forall\, j$, define

$$keep\_v^j := \bigwedge_{i=1}^{I}(E^i.next\_v^j = -1)$$

3. Define globally failure propagation (Faa) in the synchronization of EFAs.
4. $\forall\, v^j, l^i, Fa$, define globally total combined transition relations.

$$\bigwedge_{i=1}^{I}(next(l^i) = E^i.next\_l)\ \&\ \bigwedge_{j=1}^{J}(next(v^j) = S^I.next\_v^j)\ \&\ (next(Fa) = Faa)$$

V. **MODULE main**

1. $\forall\, i, j, k$, define l^{ki}, v^{kj}, F^k, ($k = 1,2$), as well as events and initial values.
2. Define two sub-modules Etot1 and Etot2. $\forall\, k$,

$$Etot^k = EFAtot(l^{k1},\ldots,l^{kI},v^{k1},\ldots,v^{kJ},F^k,e,int\_leav,k)$$

3. Define globally total transition relation for all combined locations and fault states.

$$\bigwedge_{i,k}(next(l^{ki}) = Etot^k.E^i.next\_l) \& \bigwedge_{k}(next(F^k) = Etot^k.Faa)$$

4. Verify the model by the CTL specification
   $AG\,AF\ (f_i^1 = f_i^2)$.

6.5 Conclusion

This chapter was started with a NuSMV code which implemented diagnosability analysis on an ordinary automaton with two different types of implementation, one implementing the whole code in one module, and the other one implementing it in two modules. Moreover, synchronization on an arbitrary number of nondeterministic EFAs was implemented using NuSMV code. Also, diagnosability analysis was implemented using NuSMV code where a diagnosability test was performed on the synchronization of two EFAs.

Finally, two algorithms based on the implementation were introduced, where one was on the synchronization of EFAs and the other one was on diagnosability test on the synchronization of EFAs.

7

CONCLUSION

This thesis is on diagnosability notion and failure diagnosis of discrete event systems. It starts with the definition of ordinary automata and formal languages where some fundamental notions for this work such as prefix closure, projection and inverse projection are illustrated. These notions are the building blocks of the diagnosability test algorithm and the online failure diagnosis algorithm, which were illustrated with examples through the text. Furthermore, to increase the compactness of the model EFA (Extended Finite Automaton) is expressed with condition relations augmented to it. To be more general the implemented EFAs are nondeterministic EFAs.

In this perspective, a diagnosability test algorithm is applied on the synchronization of an arbitrary number of nondeterministic EFAs. For this purpose, the nondeterministic EFAs are synchronized and then the diagnosability test is performed on the synchronization. For diagnosability verification on the synchronization of EFAs, two CTL specifications are proposed which are implemented in the NuSMV model-checker software tool.

The advantage of this work is that the compactness of EFAs along with the augmented condition relations and the nondeterminism help us to represent large systems in a compact and understandable model. Moreover, these models are more similar to real industrial systems where it is very crucial to verify the diagnosability of interacting systems and be able to diagnose failure occurrences while systems are operating.

Bibliography

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

[2] M. Sampath, R. Sengupta, S. Lafortune, K. Sinnamohideen, D. Teneketzis, Failure diagnosis using discrete-event models, IEEE Transactions on Control Systems Technology 4 (2) (1996) 105 –124.

[3] M. Sampath, S. Lafortune, D. Teneketzis, Active diagnosis of discrete-event systems, IEEE Transactions on Automatic Control 43 (7) (1998) 908 –929.

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

[5] Z. Huang, S. Bhattacharyya, R. Kumar, S. Jiang, V. Chandra, Diagnosis of discrete-event systems in rules-based model using first-order linear temporal logic, Asian Journal of Control (2008) 1–9.

[6] M. Cabasino, A. Giua, C. Seatzu, A. Solinas, K. Zedda, Fault diagnosis of an abs system using petri nets, in: IEEE Conference on Automation Science and Engineering, 2011, pp. 594 –599.

[7] M. Cabasino, A. Giua, S. Lafortune, C. Seatzu, Diagnosability analysis of unbounded petri nets, in: 48th IEEE Conference on Decision and Control, 2009, pp. 1267 –1272.

[8] M. Skoldstam, K. Akesson, M. Fabian, Modeling of discrete event systems using finite automata with variables, in: 46th IEEE Conference on Decision and Control, 2007, pp. 3387–3392.

[9] C. G. Cassandras, S. Lafortune, Introduction to discrete event systems, Springer Science+Business Media, 2008.

[10] B. Lennartson, Lecture notes on introduction to discrete event systems, Department of signals and systems, Chalmers University of Technology, 2009.

[11] S. Lafortune, D. Teneketzis, M. Sampath, R. Sengupta, K. Sinnamohideen, Failure diagnosis of dynamic systems: an approach based on discrete event systems, in: American Control Conference, Vol. 3, 2001, pp. 2058 –2071.

[12] M. Huth, M. Ryan, Logic in Computer Science, Modelling and reasoning about systems, Cambridge University Press, 2009.

[13] A. Cimatti, E. M. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, Nusmv 2: An opensourse tool for symbolic model checking, in: International Conference on Computer-Aided Verification, Vol. 2404/2002, 2002, pp. 359–364.

[14] P. Arcaini, A. Gargantini, E. Riccobene, A model advisor for nusmv specifications, Innovations in Systems and Software Engineering 7 (2) (2011) 97–107.