2003

RULES BASED MODELING OF DISCRETE EVENT SYSTEMS WITH FAULTS AND THEIR DIAGNOSIS

Zhongdong Huang
University of Kentucky, zhdhuang@engr.uky.edu

ABSTRACT OF DISSERTATION

Zhongdong Huang

The Graduate School
University of Kentucky
2003

RULES BASED MODELING OF DISCRETE EVENT SYSTEMS WITH FAULTS AND THEIR DIAGNOSIS

ABSTRACT OF DISSERTATION

A dissertation submitted in partial fulfillment of the
requirements for the degree of Doctor of Philosophy in the
College of Engineering at the University of Kentucky

By
Zhongdong Huang
Department of Electrical and Computer Engineering
University of Kentucky, Lexington, Kentucky
Director: Dr. Ratnesh Kumar, Associate Professor
Department of Electrical and Computer Engineering
Iowa State University, Ames, Iowa
2003

ABSTRACT OF DISSERTATION

RULES BASED MODELING OF DISCRETE EVENT SYSTEMS WITH FAULTS AND THEIR DIAGNOSIS

Failure diagnosis in large and complex systems is a critical task. In the realm of discrete event systems, Sampath *et al.* proposed a language based failure diagnosis approach. They introduced the diagnosability for discrete event systems and gave a method for testing the diagnosability by first constructing a diagnoser for the system. The complexity of this method of testing diagnosability is exponential in the number of states of the system and doubly exponential in the number of failure types. In this thesis, we give an algorithm for testing diagnosability that does not construct a diagnoser for the system, and its complexity is of 4th order in the number of states of the system and linear in the number of the failure types.

In this dissertation we also study diagnosis of discrete event systems (DESs) modeled in the rule-based modeling formalism introduced in [12] to model failure-prone systems. The results have been represented in [43].

An attractive feature of rule-based model is it's compactness (size is polynomial in number of signals). A motivation for the work presented is to develop failure diagnosis techniques that are able to exploit this compactness. In this regard, we develop symbolic techniques for testing diagnosability and computing a diagnoser. Diagnosability test is shown to be an instance of 1^{st} order temporal logic model-checking. An on-line algorithm for diagnoser

synthesis is obtained by using predicates and predicate transformers.

We demonstrate our approach by applying it to modeling and diagnosis of a part of the assembly-line. When the system is found to be not diagnosable, we use sensor refinement and sensor augmentation to make the system diagnosable. In this dissertation, a controller is also extracted from the maximally permissive supervisor for the purpose of implementing the control by selecting, when possible, only one controllable event from among the ones allowed by the supervisor for the assembly line in automaton models.

KEYWORDS: Discrete event system, Rules based model, Diagnosability, Diagnoser, First order temporal logic, Model checking.

Zhongdong Huang

December 2, 2003

RULES BASED MODELING OF DISCRETE EVENT SYSTEMS WITH FAULTS AND THEIR DIAGNOSIS

By

Zhongdong Huang

Ratnesh Kumar

Director of Dissertation

William T. Smith

Director of Graduate Studies

RULES FOR THE USE OF DISSERTATIONS

Unpublished dissertations submitted for the Doctor's degree and deposited in the University of Kentucky Library are as a rule open for inspection, but are to be used only with due regard to the rights of the authors. Bibliographical references may be noted, but quotations or summaries of parts may be published only with the permission of the author, and with the usual scholarly acknowledgments.

Extensive copying or publication of the dissertation in whole or in part requires also the consent of the Dean of The Graduate School of the University of Kentucky.

DISSERTATION

Zhongdong Huang

The Graduate School
University of Kentucky
2003

RULES BASED MODELING OF DISCRETE EVENT SYSTEMS WITH FAULTS AND THEIR DIAGNOSIS

DISSERTATION

A dissertation submitted in partial fulfillment of the
requirements for the degree of Doctor of Philosophy in the
College of Engineering at the University of Kentucky

By
Zhongdong Huang
Department of Electrical and Computer Engineering
University of Kentucky, Lexington, Kentucky
Director: Dr. Ratnesh Kumar, Associate Professor
Department of Electrical and Computer Engineering
Iowa State University, Ames, Iowa
2003

To

Chunfeng, Zijin and Jennifer

and

those whose presence brighten my life
and

in loving memory of days struggling for better life

Acknowledgments

I would like to express my gratitude to Dr Ratnesh Kumar whose help, encouragement, patience have been instrumental in guiding this thesis. Without him, there would not be this dissertation existing.

To my parents, Mr. Zhensheng Huang and Mrs. Donge Qin, who passed on their love by educating me as much as they could.

To my Wife, Mrs. Chunfeng Yang, for her steady encouragement and confidence in me.

To my son, Zijin Huang and my daughter, Jennifer Huang, who brought tons of joy to my life and also made my life a little bit hard.

I would like to thank my sisters: Guixiang, zhonghua, Qinghua and Xuehua for their support. I would like to thank my parents-in-law, Mr. Chenglin Yang and Mrs. Yuqin Wang for their understanging and support.

I would like to thank my colleagues - Dr. Vigyan Chandra and Dr. Shengbing Jiang, for besides being wonderful friends, two teachers who really valued my learning experience, in the research that I collaborated with them.

I would like to thank Dr. Larry Holloway, Dr. Yuming Zhang, Dr. Jon Yingling, Dr. Joseph Sottile, and Dr. D. Manivannan for serving as my advisory committee. Also to Dr. William Smith for helping smoothen out administrative matters.

I also would like to thank Mr. Kevin Pippen, President of Automation Authority Inc., who offered me an internship, which opened my career door and his great help during the period when I worked in his company.

The research was supported in part by National Science Foundation under the grants NSF-ECS-9709796, and NSF-ECS-0099851 a DoD-EPSCoR grant from the Office of Naval Research under the grant N000140110621, a KY-EPSCoR grant, and a Kenucky Research Challenge Trust Fund Grant.

Contents

List of Figures

List of Files

huang.pdf	798KB

Chapter 1

Modeling and Diagnosis of DESs

1.1 Introduction

"Systems diagnostics is a systems-oriented problem-solving-methodology able both to identify the operational (health) status and to select or design the appropriate remedial or corrective action (therapy) that would most efficiently restore the functioning of the diagnosed dynamics system to the desired level. It includes a collection of diagnostic methods, models, concepts, principles and approaches that permit us to detect and identify the functional, structural, organizational and behavioral malfunctions, breakdowns, disorders or deficiencies in various dynamic systems."[60]

"An on-going task in engineering is to increase the reliability, availability and safety of technical processes."[45]

These quotes point to the importance of failure diagnosis and recovery. This is the topic we explore for discrete event systems.

Whether it refers to medicine, where human subjects are involved, or to engineering, where technical systems are dealt with, diagnosis is the task of finding the cause of a misbehavior. According to Webster Dictionary the meaning of the term diagnosis is defined as the act or process of deciding the nature of a diseased condition by examinations, as well as a careful investigation of facts to determine the nature of a thing. This definition bears semblance to the mathematical approaches to the study of diagnosis [60][90]. The term diagnosis has Greek origin: "dia gignoskein", that means "knowing the difference". That is, diagnosis involves the act of distinguishing one case from another, of separating a relevant item from the general context. In that sense, diagnosis is closely related to categorization,

to the act of labeling or classifying.

In literature there may be found several definitions for the term fault. The Webster dictionary defines the term fault as a defect in quality or constitution. Within the engineering community a fault is regarded as physical condition that causes a device, a component or, an element to fail to perform in a required manner. The Reliability, Availability and Maintainability (RAM) Dictionary defines the fault as an accidental condition that causes a previously functional unit to fail to perform its required function. In the same source fault is regarded as an immediate cause of a failure (often classified based on duration, extent, value and whether the cause was physical or humane). In turn, a failure is defined as an event that makes equipment deviate from specified limits of useful performance, or that terminates the ability of a unit's material or structure from performing its required function. Due to failure, a malfunction occurs, which is defined to be the inability of a system or system component to perform a required function within specified limits (inability to meet or conform to a specified requirement). The discrepancy between a computed, observed or measured value or condition/requirement, and true, specified or theoretically correct value or condition/requirement is termed error.

In our work, "failure" and "fault" mean the same and are used interchangeably. They refer to a non-permitted deviation in the behavior of the system (or a component of the system) for a bounded or unbounded period of time. A stuck-close valve, decrease in the efficiency of a heat exchanger, abnormal bias in the output of a sensor, and leakage in pipelines are examples of failure. If after the occurrence of a failure, the system remains in the faulty condition indefinitely, then the failure is called permanent. Otherwise, it is non-permanent or transient. A broken shaft in a motor is an example of permanent failure, and a loose wire could be the source of a transient failure in an electrical system.

A typical fault diagnosis system uses the outputs of the sensors of the system to detect the failure and (if necessary) isolate (locate) the source of failure. Once a failure is detected, a decision has to be made as to how it should be recovered from. It is necessary to detect and further isolate the source of failure to be able to perform failure recovery effectively. So, a failure recovery problem involves a failure-diagnosis problem.

1.2 Failure diagnosis: Motivation and Approaches

There are three major factors that motivate research on failure diagnosis:

1. failures are inevitable;

2. failure diagnosis is needed for recovery; and

3. failure diagnosis is a complex task.

Failures are inevitable in today's complex industrial environment. Given the complex interactions between components, sub-systems, and processes, a system failure can well be considered to be a "normal" occurrence [85], or an inherent characteristic of most industrial systems.

If a failure is detected late, it might "spread", and cause unnecessary operational disturbances, and even material and personal damage. Timely and accurate detection of these failures may prevent the cascaded effect that simple failures produce, resulting in system-wide breakdowns and major accidents.

Also, a failure may well be easy to fix, but hard to find. A skilled operator can often quickly isolate the failure in systems (s)he is familiar with, but as systems grow more complex, this manual diagnosis becomes more difficult, and specialists capable of performing it are more expensive to train. It is therefore of interest to have, if not totally automated, sophisticated diagnosis tool for complex systems. In view of the above mentioned factors, one can easily appreciate automated mechanisms for the timely and accurate diagnosis of failures. Indeed, this need is well understood and appreciated both in industry and in academia. A great deal of research effort has been and is being spent in the design and development of automated diagnostic systems; and a variety of schemes, differing both in their theoretical framework and in their design and implementation philosophy, have been proposed.

A traditional approach to diagnosis used in many industrial systems is simple limit checking of signal values and predefined threshold logic. For example, if a sensor value leaves its normal range, an alarm is generated. Due to cascaded effect of failure propagation in the system, often a number of alarms are set off and then the problem is to isolate the root cause. Common methods used are fault trees and expert systems, although these methods are not restricted only to alarm analysis.

1.2.1 Non-model based approaches

In this section we briefly describe some approaches with the common feature that they explicitly associate a *known* failure with an *observed* misbehavior. The kind of pattern-matching methods these approaches use are often called associative.

Fault trees

Fault tree analysis is a widely used technique in the process control industry for reliability analysis, fault detection and isolation. The basic idea is that a failure can trigger other failures or events in the system and this can be traced back to the root cause [35][109]. A fault tree graphically represents a cause-effect relationship among the failures in the system. The root of a fault tree, the so called TOP event, is a system failure. The leaves of the tree are the possibly contributing atomic events or basic faults, and inner nodes are AND- and OR- type. Sets of events that trigger the top event are computed using cut sets and minimal cut sets. By assigning probabilities to the atomic events, a failure probability can be found.

Fault tree construction is laborious and error prone, and much work has been done on computer assisted and automatic fault tree construction, see for instance [58][25][110].

Expert systems

A popular method for diagnosis and supervision of complex systems has been the use of expert systems, often in conjunction with fault tree structures, see, e.g., [101][107]. Expert systems are especially well suited for systems that are difficult to model, with complex interactions between and within components. Domain experts have heuristic knowledge of the system and of how symptoms relate to faults. In traditional expert systems, this knowledge is represented in a rule-base and used in conjunction with an inference engine.

This heuristic approach has several drawbacks. Acquiring knowledge from experts is difficult and time consuming, and for new systems a considerable amount of time might elapse before enough knowledge has accumulated to make reliable diagnosis possible.

Chronicles

An expert system-like approach to diagnosis of dynamic systems using temporal information, is the real-time situation recognition method described in [8][73]. For system under consideration, a set of events, obtainable for instance by signal processing, is defined. A number of situations, that are considered desirable to recognize, are characterized with these events, and also the temporal constraints among the events. Such a characterization of a situation is called a chronicle and can correspond both to correct operation and failures of the system. A situation recognizer is then fed with observed and time-stamped events in real-time and can notify an operator for example when a failure situation has occurred or is

under development. Of course only known situation can be recognized.

Fuzzy logic based approach

Tsukamoto and Terano used a set of fuzzy relational inequalities in order to describe the intensity of the causal deterministic relationships existing between faults (as causes) and the determined symptoms (as effects) [106]. Since then this idea has been successfully applied for diagnosis of complex industrial processes, based on subjectively observed symptoms, see [27][51]. Sanchez has built up a "symptoms-faults" fuzzy relational mapping by directly encoding expert medical diagnostic knowledge [100]. As a result, A fuzzy rule based fault detection and diagnosis system can be developed by combining fuzzy linguistic rules with non-fuzzy numerical data [119]. The design of the fuzzy reference set, inference mechanism, and signal coding and decoding policies are dependent on the problem background.

Bayesian networks based approach

Bayesian network has been referred to by different names in the literature: Bayes belief net [83], causal probabilistic network [94], causal networks [61], probabilistic causal networks [19], and influence diagrams [40]. At the qualitative level, it is a graph where the nodes represent domain objects and the arcs between nodes represent causal relationships between objects, which is then modeled by conditional probabilities, and processed via the Bayes-Laplace formula [24]. Bayesian network can be used to study the diagnosis problem. It computes the probabilistic evidence of the unobserved part of the domain given that a part of the domain (the symptoms variables) has known or assumed values (in form of probability distributions).

This approach has its own drawbacks. The priori necessary probabilistic information is not always available and event independence does not always hold. Computation complexity is almost intractable [29].

Neural network

A neural network being applied for fault detection and diagnosis is developed through 'learning', i.e. 'learning from examples', even if one does not know the 'if-then' kind of linguistic rules or process principle in detail [53]. The design of neural network architectures and learning algorithms depends on the forms of learning examples. The key task is to

establish a neural network with an associative memory which can classify the input space into a number of fault related domains. The learning algorithm designed mainly depends on the informative form and content in the training samples; generally, 'BP-learning', 'reinforcement learning', or 'self-organized learning' can be used. The fault diagnosis or decision making for control actions is a more complicated problem, particularly for a high dimensional, nonlinear, and uncertain system.

1.2.2 Model based approaches

The basic idea of all model based diagnosis is to compare observations of the real system with the predictions from a model. In the case of a fault, a discrepancy between the actual observed behavior and the predicted behavior arises. This discrepancy can then be used to detect, isolate and identify the fault depending on the type of model and methods used. Several approaches to model based diagnosis with model paradigms ranging from differential equations to qualitative behavior models have appeared in the literature and proven useful in practice.

Analytical redundancy methods

In the control system community, the most common class of model based diagnosis method proposed is the analytical redundancy method, see for instance [114][81][28][74] [92] and references therein. These methods are based on the fact that observed signals from the system, such as sensor measurement and control signals, contain information regarding the system state. The desired state information can be extracted with the help of a good differential equation model of the system, often obtained by physical modeling, or system identification [70].

In general, these methods consist of two steps. First, residuals are generated by comparing observed signals from the system with predicted values. These residuals are usually designed to be zero if no fault is present. Second, a fault detection and isolation step is employed, using these residuals as input. One of the main problems with this approach is the difficulty in acquiring good enough models. The demands on the accuracy of the models are usually higher than for control design, since the residual generator works open-loop. Robust methods for residual generation has received considerable attention in recent years and is an active research area, see, e.g., [81][82].

Recently, there have also been some efforts towards formally demonstrating the connections between the analytical redundancy methods and the model-based approaches in AI briefly described in [74].

Model based approaches in AI

In artificial intelligence, model-based diagnosis from first principles, is pioneered by Reiter [91]. The basic idea is to predict the behavior of the system using behavioral and structural models of the system and its components and compare it with observations of the actual behavior of the real system. Two main characterizations of model based diagnosis exists in the literature, consistency based diagnosis [91] and adductive diagnosis [87]. The former needs a model of the correct behavior only, and a diagnosis is a set of components that when assumed non-correct, makes the predicted behavior consistent with the observed. The adductive approach must have models of the faulty behavior. A diagnosis is a set of component faults that is not only consistent with, but also explicitly predicts the observed behavior. A good introduction to model-based diagnosis is the Chapter 2 of [30], and for a comprehensive collection of literature, see [34].

Model based diagnosis in AI is mainly aimed at static systems and especially at troubleshooting combinatorial digital circuits. The extension of the methodology to dynamic and time-varying systems remains an active research area, see., [30][17][18][113].

Template based approach

A method for fault monitoring of automated manufacturing systems using the timing and sequencing of events has recently been developed by Holloway et al. [39][37][21][78]. So called time templates or condition templates are used to specify the expected, correct timing and sequencing of events. Templates are designed to be easily implemented on distributed architectures and are used to detect deviations from the correct behavior. Specific faults can not be diagnosed. The method is capable of monitoring event sequences corresponding to an arbitrary number of concurrent timed automata, and a timed automata specification can be automatically translated to templates [39][78]. The manual construction of templates is not feasible when timed automata model is not known, and work has been done on identifying templates from observations of sequences of events [21].

A state based DES approach to diagnosis failure

A discrete event system approach for diagnosis is proposed by Lin [67] and further treated in [5]. The system is modeled as a finite state machine, where the states of the machine describe conditions of the components. The only dynamics in the FSM is that the system can transit from normal to faulty, and no normal behavior is modeled. The observations (sensor readings) are included as a mapping from the fault states to certain observable events (tests). Diagnosing a fault is equivalent to identifying which state or set of states the system belongs to. Off-line and on-line diagnosis are treated separately.

In off-line diagnosis the system is thought to be in a test-bed, where the system does not change stage unless it is forced to while under diagnosis. Test can be performed, and the outcome of the tests are called observable events. The off-line diagnosability of the system is analyzed with respect to a fault partition and the set of observable events (i.e., tests). Since the system is in a test-bed, the order in which the tests are performed does not affect the diagnosability.

In on-line diagnosis, the system is in normal operation, and hence it can change state uncontrollably. The system is assumed to be partially observable via an output map. Given a state (fault) partition, the system is defined to be on- line diagnosable if there exists sequence of commands so that the state of the system, up to the fault partition, can be decided from the output map. An algorithm for computing such a sequence is given in [67].

Formal language based DES approach

Sampath et al. has proposed a formal language framework for studying diagnosability properties of un-timed discrete event systems [98][97][95]. The approach is closely related to the Ramage-Wonham framework for supervisory control of DES [89]. The method has been applied to HVAC- systems (heating, ventilation and air-condition), but is applicable to all systems that at some level can be meaningfully modeled as discrete event systems. The main features of the approach is methods for modeling the normal and faulty behavior of systems as DES, implementation of online passive diagnosis, and analysis of diagnosability properties.

Both the normal and faulty behavior of the system to be diagnosed is modeled with a finite state machine. Components are modeled individually and faults in the components are modeled as unobservable events. The system model is then put together with the usual synchronous composition [89]. To include sensor information in the model, global sensor

maps from the states of the model to sensor readings are constructed and then included in the event labels. The diagnostic problem is then to infer about past occurrences of unobservable fault events from the observable events. A diagnoser that gives a state and fault estimation of the system after occurrence of each observable event is constructed from the model. The diagnoser hence is an extended observer.

One of the main features of this approach is the ability to analyze diagnosability properties. Diagnosability is, the ability to detect and isolate an occurred fault with finite delay using the observable events. Isolation is performed with respect to a fault partition.

To demand that a fault should always be diagnosed with finite delay is rather strong, and very few systems can fulfill it. Therefore the notion of I-diagnosability is introduced, which means that a fault should be detected and, up to a given partition, isolated with finite delay after a so called indicator event has occurred. Stated in other words, the system has to be excited enough for the fault to be diagnosed.

If a model of a system is not diagnosable (I-diagnosable), the authors identify two means of making it diagnosable: i) introduce more sensors and ii) design the controller so that the faulty behavior is excited and can be detected. The theory is set in a formal language framework. In [95] and [96], the authors present a method for designing controllers in the RW-framework that make the system diagnosable. Chen and Provan have extended the approach to timed discrete event systems [14].

1.2.3 Temporal logic based approach

Temporal logic was originally developed [42] for investigation the manner in which temporal operators are used in natural language arguments. It provides a formal way of qualitatively describing and reasoning about how the truth values of assertions change over time. In [86], first argued that temporal logic is appropriate for reasoning about nonterminating concurrent programs such as operating systems and network communication protocols. Now temporal logic is a widely active area of research. It has been used or proposed for use in virtually all aspects of concurrent program design, including specification, verification, and mechanical program synthesis.

For the supervisory control and failure diagnosis problem, temporal logic provides an effective means of specification. In most cases, the translation of a simple natural language specification into temporal logic one is quite straightforward.

In [47], linear time temporal logic (LTL) is used to express fault specifications for the

failure diagnosis of DESs. Diagnosability of DESs is defined in the temporal logic setting. The problem of testing the diagnosability is reduced to that of model checking. Algorithms for the test of diagnosability and the synthesis of a diagnoser are obtained. The problem of the failure diagnosis of repeated faults is studied in [49].

This concludes our discussion on approaches to failure diagnosis that have appeared in the literature. Each of the above methods possesses certain advantages and disadvantages, and is best applicable under specific circumstances. Which of these approaches one selects for a given system depends not only on the characteristics of the system and the knowledge available about the system but also on the nature of the failures one wants to diagnose. With this background on the importance of the failure diagnosis problem and the various methodologies to solve this problem, we now proceed on to discussion related to our research.

1.3 Focus of the Research

1.3.1 Application of supervisory control to an assembly line

Modern machining and assembly facilities require a great deal of operational and structural flexibility owing to the rapidly changing manufacturing environments in which they exist. The design of controllers for such systems is often an error prone task, since intuitive methods rather than formal techniques continue to be used. Altering any existing control code in order to accommodate for changes in the system or the control objective, necessitates extensive verification to establish whether the control code actually implements the desired specifications. The theory of supervisory control meets the need of designing the controllers formally, guaranteeing that the behavior of the controlled system meets the desired control specifications, while providing maximally permissible controlled behavior. No additional testing is required to check the correctness of the code as the technique used is guaranteed to enforce the specifications. In order to demonstrate the usefulness of the supervisory control theory (SCT) in manufacturing systems, an educational test-bed that simulates an automated car assembly line has been built using LEGO blocks. Finite automata are used for modeling operations of the assembly line, and for the specifications that accomplish the task of successfully completing the assembly repeatedly. Using a set of desired safety and progress specifications for assembly, we use supervisory control techniques for automatically deriving a supervisor that enforces the specifications while offering the maximum flexibility of assembly. Subsequently a controller is extracted from the maximally permissive super-

visor for the purpose of implementing the control by selecting, when possible, at most one controllable event from among the ones allowed by the supervisor.

1.3.2 Polynomial test for diagnosis

Failure diagnosis in large and complex systems is a critical task. In the realm of discrete event systems, Sampath *et al.* proposed a language based failure diagnosis approach. They introduced the diagnosability for discrete event systems and gave a method for testing the diagnosability by first constructing a diagnoser for the system. The complexity of this method of testing diagnosability is exponential in the number of states of the system and doubly exponential in the number of failure types. In this dissertation, we give an algorithm for testing diagnosability that does not construct a diagnoser for the system, and its complexity is of 4^{th} order in the number of states of the system, 2^{nd} order when system model is deterministic, and linear in the number of the failure types.

1.3.3 Modeling discrete event systems with faults using a rules based modeling formalism

Obtaining accurate models of systems which are prone to failures and breakdowns is a difficult task. We present a methodology which makes the task of modeling failure prone discrete event systems (DESs) considerably less cumbersome, less error prone, and more user-friendly. Diagnosis of failures in discrete event systems, as proposed by Sampath *et al.* [98], makes use of a language based formulation and requires equivalent automata (state machine) models of the system. The task of obtaining such a model for DESs is non-trivial for most practical systems, owing to the fact that the number of states in the commonly used automata models is exponential in the number of signals and faults. In contrast a model of a discrete event system, in the rules based modeling formalism proposed in [12], is of size polynomial in the number of signals and faults. In order to model failures, the signals set consisting of actuators and sensors is enlarged to include binary valued fault signals, each signal representing either a non-faulty or a faulty state of a certain type. Addition of new fault signals requires introduction of new rules for the added fault signal events, and also modification of the existing rules for non-fault events.

1.3.4 Diagnosis of discrete event systems in rules based model using first-order logic

We study diagnosis of discrete event systems (DESs) modeled in the rule-based modeling formalism to model failure-prone systems. An attractive feature of rule-based model is it's compactness (size is polynomial in number of signals). A motivation for the work presented is to develop failure diagnosis techniques that are able to exploit this compactness. In this regard, we develop symbolic techniques for testing diagnosability and computing a diagnoser. Diagnosability test is shown to be an instance of 1^{st} order temporal logic model-checking. An on-line algorithm for diagnoser synthesis is obtained by using predicates and predicate transformers.

1.3.5 Rules based modeling of an assembly line and its diagnosis

We study diagnosis of an assembly-line [11] that is modeled in the rules-based modeling formalism. In order to demonstrate the usefulness of the rules-based model and of diagnosis techniques based on such a model in manufacturing systems, an educational test-bed that simulates an automated car assembly-line built using LEGO® blocks is being employed. In this dissertation we provide a rules-based model of the assembly-line. Next we demonstrate the diagnosis technique for a rules-based model, that is based on 1^{st}-order temporal logic model checking [44], by applying it to a part of the assembly-line. When the system is found to be not diagnosable, we use sensor refinement and sensor augmentation to make the system diagnosable.

1.4 Organization of the Dissertation

In *Chapter 2* the DESs notation used in the dissertation is explained. We used supervisory control theory to obtain a supervisor for the miniature factory built out of LEGO blocks at the University of Kentucky.

In *Chapter 3* a polynomial time algorithm for diagnosability of Discrete Event Systems is provided.

In *Chapter 4* we modeled Discrete Event Systems with faults using a rules-based modeling formalism.

In *Chapter 5* we discussed diagnosis of Discrete Event Systems in rules based model using First-order logic and present an on-line algorithm for diagnoser synthesis.

In *Chapter 6* we model a failure prone assembly line in rules based formalism and discussed its diagnosis.

Chapter 7 provides conclusions and points out possible directions for future work.

Appendix contains the NuSMV program for mouse-cat example. NuSMV is a software tool for performing model-checking tests that we use to verify diagnosability.

Chapter 2

Automated Control Synthesis for an Assembly Line using Discrete Event System Control Theory

2.1 Introduction

A significant increase in the level of automation [33] and system complexity has, at the turn of the century, necessitated the use of methods which do not rely on informal, intuitive or heuristically designed control programs for real time computers and programmable controllers [72]. Present day controller implementations, even those using specialized logic-based control languages like Grafcet [1], remain largely based on the expertise and experience of the designer, rather than on formal control design approaches that have been developed for discrete event control in recent years. As a result there is no way of ensuring that control specifications will be met every time in the controlled system. Supervisory control theory (SCT) proposed by Ramadge-Wonham [88] is particularly well suited for the task of controller design since the resulting supervisor is always guaranteed to meet the control specifications. This theory is applicable to any system which evolves in response to events that are spontaneous, instantaneous, asynchronous and thus discrete in nature. Such systems are classified as discrete event system (DES) and have been examined in detail [54, 88].

A DES to be controlled, also called a *plant*, is modeled by a finite state machine (FSM) and can equivalently be described by a language model. The specifications which express the constraints that one wishes to impose on the plant's behavior are modeled as formal languages

as well. A supervisor exercises control over the plant by dynamically disallowing a minimal set of controllable events so as to achieve the desired specifications. Thus the supervisor is designed to be *maximally permissive* as in the supervisory control theory. A controller is extracted out of the supervisor by selecting, when possible, at most one controllable event from among the ones allowed by the supervisor. This is illustrated in Figure 2.1. The steps

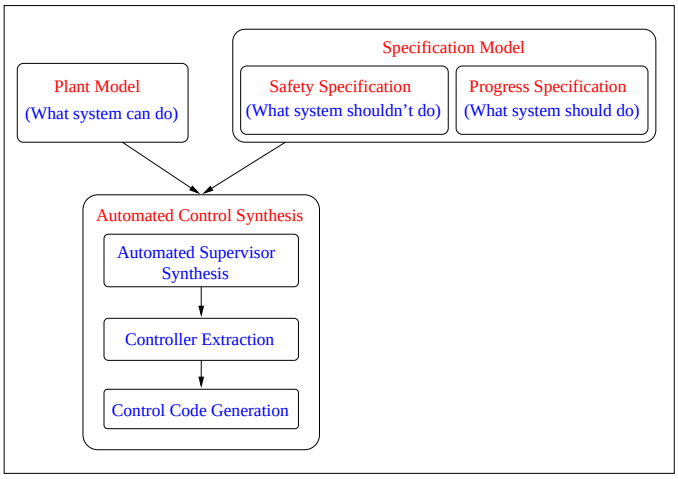

Figure 2.1: Discrete event control theory based automated control synthesis

to be followed for designing control programs using supervisory control theory are:

- FSM models of the system to be controlled.
- FSM models of the safety and progress control specifications of the system.
- Use of supervisory control theory to obtain the maximally permissive supervisor for the system.
- Extraction of a controller from the supervisor, which permits at most one controllable event to be enabled at each state.
- Translation of the controller into control code or PLC.

Some of the advantages of using SCT for automated control synthesis are as follows:

- Any change in the control specifications can be rapidly translated into executable control code using the procedure shown in Figure 2.1. The new control specification is modeled as an FSM and using SCT a new controller can be synthesized.

- No testing of the control code obtained using SCT is needed, since the method of construction of these controllers guarantees their correctness. This eliminates the time spent in checking the code for incorrect or incomplete operation sequences.
- Expensive mechanical safeguards which are installed in the system can be reduced owing to the fact that unsafe behavior in the system will be avoided when the controllers are designed using a FSM model based approach provided by SCT.
- The FSMs used for modeling the system and the specifications are intuitive and simple to construct for moderate size systems.
- Since many manufacturing systems operate in similar configurations, the models developed for one system can be altered for use in another context. Hence development time for the generation of control code in a new system can be reduced.
- The supervisor obtained from the automated control synthesis procedure shown in Figure 2.1, provides the maximum permissible ways of achieving the control objective. The designer of the system can choose which particular controller to extract out of the supervisor based on different criteria, such as minimum operation time, cost, or on the plant layout. At a later time a different controller can be chosen if the design criteria changes.
- Since the control code can be generated automatically the system designer can work at an abstracted level, not bothering about the manner or language in which the control code implementation will be done.

In this dissertation we describe a simple educational test-bed built from LEGO blocks that simulates an automated car assembly line. The objective is to demonstrate a formal way of designing a controller for a discrete event plant by applying the theory of supervisory control. This miniature assembly line performs a very simple assembly of the roof and the chassis. The two parts are transported to the press section from their respective initial sections, where a vertical press operation assembles the two parts, and finally the assembled part exits the assembly line through the unloading section. A transporter links the chassis, roof, press, and unloading sections.

We present FSM models of each of the individual sections, the composition of which is the entire plant model. The number of states in transporter, chassis, roof, press, and unloading sections is 21, 21, 30, 23, and 5 respectively, which implies a total of around 8×10^5 states

for the entire system. We also provide safety and progress specification models, where safety specification is needed for the safe operation of the system whereas the progress specification is needed to achieve the task of assembly. The safety specification is a conjunct of sixteen sub-specifications, divided into a set of six "global" ones and a set of ten "local" ones. There is a single progress specification which is obtained by combining four different sub-tasks. The overall specification is the conjunct of the safety and progress.

The set of input events which control the motor actions forms the set of controllable events, whereas the set of output events which are generated by the sensors forms the uncontrollable events set. We do not model failures, and all events are considered to be observable.

Using the supervisory control theory we obtain the maximally permissive supervisor for the miniature assembly line that enforces the overall specification. This turns out to be the automaton represented by the overall specification itself, since the overall specification is found to be controllable [54, 88]. A controller is extracted out of the supervisor as described above, and then the controller is translated into specific code understood by the LEGO Dacta control software.

While setting up the miniature LEGO factory, the one built at the University of Massachusetts, [10] in 1995 served as a prototype. In another allied work, a train test bed for controlling the movement of two trains which share three track loops, was built at the University of Toronto [62], in 1996. An application from the semiconductor industry, for controlling a Rapid Thermal Multiprocessor and addressing the reliable update of processing recipes, using supervisory control theory (SCT) has been done in [3]. Application of SCT to manufacturing is considered in [9] wherein movement of pallets carrying parts with different machining sequences required is controlled. Supervisory control has also been applied for designing supervisors in automated highway systems' vehicle communication protocols [41, 108], in protocol conversion [57], in feature interaction in telephony [104], in failure diagnosis of HVAC systems in [99].

This rest of the chaper is organized as follows: In section 2 the notations used for modeling DESs are introduced. Section 3 outlines the working of the LEGO factory and FSM models of the same are presented in section 4. Section 5 contains descriptions as well as the FSM models of the safety specifications, and Section 6 does the same for progress specifications. In section 7 a supervisor is synthesized from which a controller is extracted for the LEGO factory, and finally in section 8 scope for future extensions as part of ongoing research is

outlined.

2.2 Notation and Preliminaries

We use Σ to denote the finite set of events over which a DES evolves. A concatenation of finite number of events forms a *string* of events or a *trace*. A *language* is a collection of traces. Let Σ^* be the set of all strings (traces) of events of Σ including the empty string ϵ. A *language* is thus a subset of Σ^*. For a language H, the notation $\overline{H}$, called the *prefix closure* of H, is the set of all prefixes of traces in H. H is said to be *prefix closed* if $H = \overline{H}$.

Abstractly, a discrete event system can also be viewed as a 5-tuple state machine

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

where X is the set of states, Σ is the set of events, $\delta : X \times \Sigma \to X$ is the partial state transition function, $x_0 \in Q$ is the initial state, and $X_m \subseteq X$ is the set of marked or final states. The generated behavior of the discrete event system modeled by G is described by its *generated* language:

$$L(G) := \{s \in \Sigma^* | \delta(s, x_0) \text{ is defined}\},$$

where by induction the transition function has been extended from events to traces $\delta : X \times \Sigma^* \to X$. The *generated language* of G is the set of all traces that it can execute starting from its initial state. The *marked* language of G contains those generated traces which terminate in a final state and signify task completion:

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

Synchronous composition [36] of state machines is used to represent the concurrent behavior of two DESs. Given two deterministic state machines $S_1 := (X_1, \Sigma_1, \delta_1, x_{0,1}, X_{m,1})$ and $S_2 := (X_2, \Sigma_2, \delta_2, x_{0,2}, X_{m,2})$, composition of S_1 and S_2 denoted $S_1 \| S_2 := (X, \Sigma, \delta, x_0, X_m)$, is defined as: $X := X_1 \times X_2$, $\Sigma := \Sigma_1 \cup \Sigma_2$, $x_0 := (x_{0,1}, x_{0,2})$, $X_m := X_{m,1} \times X_{m,2}$, and for each $x = (x_1, x_2) \in X$ and $\sigma \in \Sigma$:

$$\delta(x, \sigma) := \begin{cases} (\delta_1(x_1, \sigma), \delta_2(x_2, \sigma)) & \text{if } \delta_1(x_1, \sigma), \delta_2(x_2, \sigma) \text{ defined}, \sigma \in \Sigma_1 \cap \Sigma_2 \\ (\delta_1(x_1, \sigma), x_2) & \text{if } \delta_1(x_1, \sigma) \text{ defined}, \sigma \in \Sigma_1 - \Sigma_2 \\ (x_1, \delta_2(x_2, \sigma)) & \text{if } \delta_2(x_2, \sigma) \text{ defined}, \sigma \in \Sigma_2 - \Sigma_1 \\ \text{undefined} & \text{otherwise} \end{cases}$$

So when S_1 and S_2 are composed, the common events occur synchronously, while the other events occur asynchronously. Note that when $\Sigma_1 = \Sigma_2 = \Sigma$, then $L(S_1 \| S_2) = L(S_1) \cap L(S_2)$ and $L_m(S_1 \| S_2) = L_m(S_1) \cap L_m(S_2)$ since all events must occur synchronously.

The set of plant events is partitioned into two disjoint sets Σ_c, the set of all *controllable* events, and Σ_u, the set of all *uncontrollable* events. A *controllable* event is one which can be allowed to occur or prevented from possibly occurring by an external agent, whereas no such control is possible for an *uncontrollable* event.

In general, a supervisor determines the set of events to be disabled after each transition based on the observed sequence of events. A supervisor, denoted S, is a map $S : L(G) \rightarrow 2^{\Sigma - \Sigma_u}$ that determines the set of events $S(s) \subseteq (\Sigma - \Sigma_u)$ to be disabled after the occurrence of trace $s \in L(G)$. Events not belonging to the set $S(s)$ remain enabled at trace s. In particular, the uncontrollable events remain enabled. A supervisor thus restricts the behavior of the plant. Since synchronous composition also results in behavior restrictions, the action of control may also be achieved by taking a synchronous composition of plant automaton G and supervisor automaton S. This is represented by the automaton $G \| S$. Since S must never prevent any feasible uncontrollable event from happening, the following should hold: $L(G\|S)\Sigma_u \cap L(G) \subseteq L(G\|S)$, in which case S is said to be Σ_u-enabling [54]. Further, S is said to be non-blocking if $L(G\|S) \subseteq \overline{L_m(G\|S)}$, i.e., if each generated trace of the controlled plant can be extended to be a marked trace of the controlled plant.

It is known from supervisory control theory [88] that given a discrete event plant G and a desired nonempty specification language $K \subseteq L_m(G)$, there exists a Σ_u-enabling and nonblocking supervisor S such that $L_m(G\|S) = K$ if and only if K is *controllable* and relative-closed with respect to G, i.e,

$$\overline{K}\Sigma_u \cap L(G) \subseteq \overline{K}, \text{ and } \overline{K} \cap L_m(G) \subseteq K.$$

Controllability means execution of an arbitrary prefix of K, say s, followed by an uncontrollable event σ; such that $s\sigma$ is possible in the plant; implies that $s\sigma$ is also in the prefix of K. This is because the occurrence of uncontrollable events cannot be prevented. Relative-closure means that any prefix of K that is marked by the plant must itself be in K. This is because the marking status of a trace is determined by the plant.

If K is controllable with respect to $L(G)$ then the synchronous composition of the automata representing K and $L(G)$ is the required supervisor, but if this is not the case. then one computes the language $K^{\uparrow}$, the *supremal controllable and relative-closed sublanguage* of K with respect to G [116, 55].

A controller we design, further restricts the behavior of the plant under maximally permissive supervision, with the property that it permits the execution of only one controllable event following each trace, whenever at at least one such event is possible following that trace. A controller is defined to be a mapping, $\mathcal{C} : \overline{K^{\uparrow}} \rightarrow \Sigma_c \cup \{\epsilon\}$:

$$\forall s \in \overline{K^{\uparrow}} : \begin{cases} \mathcal{C}(s) \neq \{\epsilon\} & \text{if } (\overline{K^{\uparrow}} \backslash s) \cap \Sigma_c \neq \Phi \\ \mathcal{C}(s) = \{\epsilon\} & \text{otherwise} \end{cases}$$

As with a supervisor, the uncontrollable events remain enabled.

2.3 Example: Control of a LEGO transporter

For the sake of illustration of SCT, we first present a simpler example of a LEGO® transporter system. In subsequent sections, the FSM modeling and supervisory control of the test-bed LEGO® factory are provided. A transporter, shown in Figure 2.2, moves between home and extended positions, crossing a number of intermediary positions. An angle sensor,

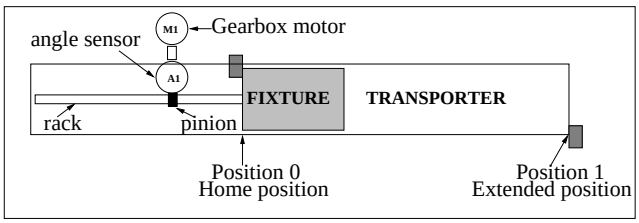

Figure 2.2: Schematic of the transporter

A1, is used to determine when the transporter is either in the extended, *f*, intermediate, *a,g*, or the home position, *l*. The forward and reverse direction motor commands are *Ifon* and *Iron* respectively, while the corresponding stop commands are *Ifof* and *Irof*.

An automaton model of the system is constructed by considering all possible sequence of events possible in the plant starting from the initial state. We assume that the initial state of the system is when all the actuators are off (*Irof, Ifof*) and the transporter is in retracted position (*l*). In the initial state the controllable events *Ifon*, and *Iron*, are possible. *Ifon* will cause the transporter to leave the home position in the forward direction and enter the intermediary position, *a*. On the other hand, the *Iron* command will not change the position of the transporter as it already is in the home position. In this way the automaton model is constructed by considering all possibile events, at all the possible states reachable from the

initial state. In the FSM models controllable transitions are indicated by a short line drawn across the transitions. Filled circles in the FSM models represent marked or acceptable behavior of the system. The FSM model of the transporter system is shown in Figure 2.3.

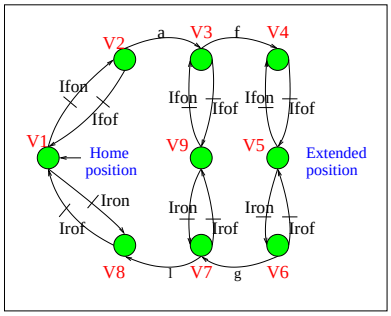

Figure 2.3: Overall FSM model of the transporter

2.3.1 Specification Models

Safety specifications are concerned with the safe operation of the plant and must be enforced regardless of what task the plant is performing. The progress specification is used to specify the specific task the plant needs to perform. Since the prefixes of safe operations must themselves be safe, safety specifications are prefix closed. In contrast, the progress specifications are non-prefix closed. The safety specifications for the model is shown in Figure 2.4, which indicate that when the transporter reaches the extended position the forward motor should no longer be kept on, and similarly when the transporter reaches the home position the reverse motor should not be kept on. The progress specification is also shown in Fig-

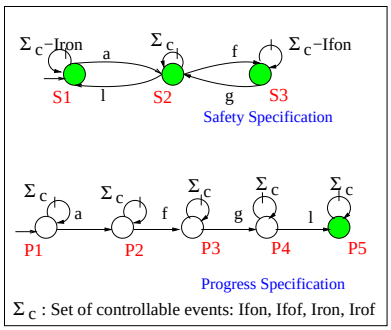

Figure 2.4: Safety and Progress specification FSM models

ure 2.4, wherein the transporter should commence movement from the home position, travel

until the extended position and return to home, whence all movement should be turned off permanently, i.e., the cycle should not be repeated.

2.3.2 Supervisor Synthesis and Controller Extraction

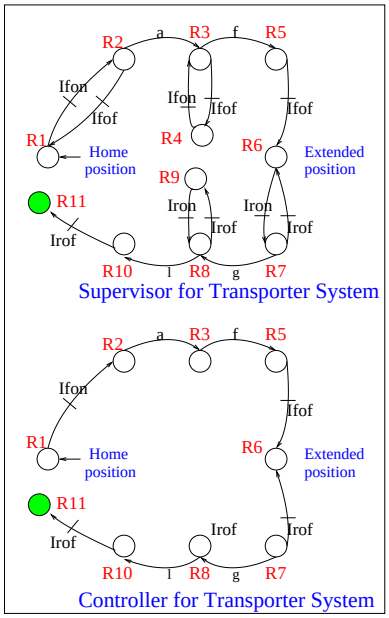

Figure 2.5: Supervisor and Controller FSM models

An overall FSM model of the specifications is first obtained by taking a synchronous composition of the safety and progress specifications, yielding a combined specification having 5 states. For computing a supervisor, we use the DES software toolkit, and this yields a supervisor for the transporter system having 11 states. The supervisor and one of the possible candidate controllers is shown in Figure 2.5. Note that at any stage of the assembly process the controller enables at most one controllable event.

2.4 Description of LEGO Assembly Line

This miniature assembly line simulates the conditions under which actual automobile assembly take place, and involves: motors which drive mechanisms which in turn cause the assembly of the roof-chassis to take place; a transporter to move the semi-finished product

through various stages of the assembly; and sensors which bring back the status of the present plant conditions to the LEGO DACTA controller.

The pieces to be assembled are shown in Figure 2.6. The miniature factory is made

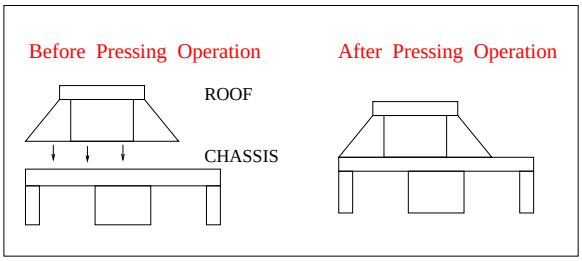

Figure 2.6: Partially assembled car

entirely out of LEGO blocks and contains 8 sensors and 9 motors, also provided by LEGO. The factory layout required for implementing this, and to provide for material handling and storage capabilities is given in Figure 2.7.

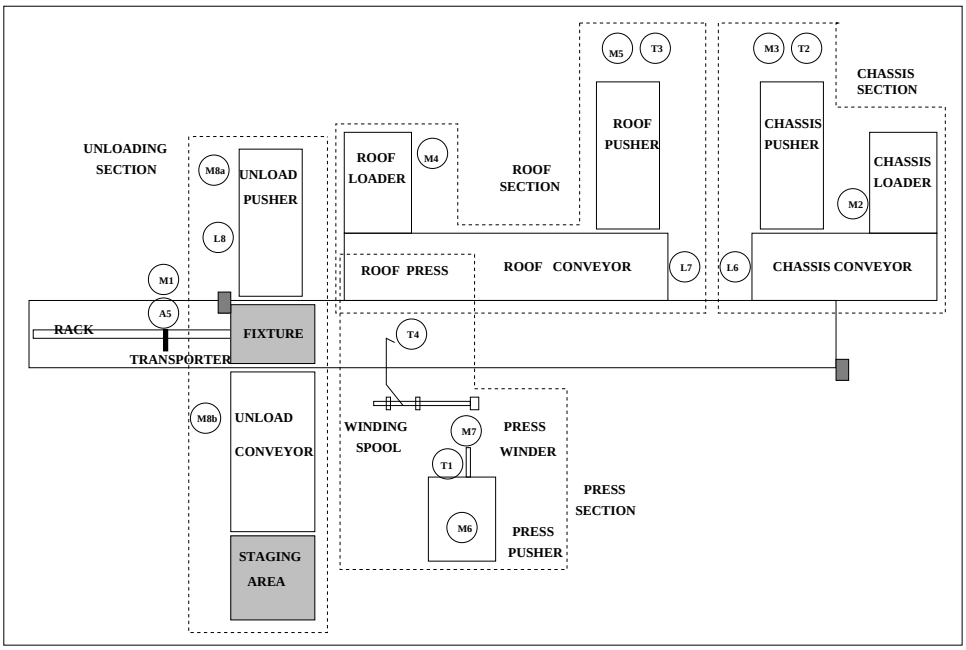

Figure 2.7: Schematic of the plant layout

The factory is controlled by one personal computer, which is interfaced with the assembly-line through a **LEGO Control Lab Interface Box**. This interface box has 8 sensor inputs, four of which accommodate *passive* (binary) sensors viz. touch; while the other four

accommodate *active* (continuous valued) sensors viz. light or angle. **Touch** sensors return a value of boolean value when depressed, indicating their logic status. **Light** sensors return the light intensity reflected into the sensor's detector, either as a percentage or as a raw number. **Angle** sensors which are always connected to a rotating axle, report either the angle in degrees or the number of revolutions turned.

In addition to sensor inputs, the interface box also provides eight motor outputs. The power and directions of the motors, as well as the overall logic of the plant can be controlled through the Control Lab Software which is written in a special version of the "Logo" programming language, called **Control lab**. Complete control of the operations involved in the assembly is done through this MS-WINDOWS based software. In our setting, control of the plant is done using LEGO commands obtained from the controller designed using supervisory control.

The capability of the system under investigation is quite extensive and we limit it by imposing certain restrictions which are as follows:

- We do not change the speed of any of the motors without first stopping them, even though this is possible through the software. Also if the direction of the motors have to be changed then they are brought to a stop prior to being enabled for rotation in the opposite direction.

- Owing to the limitations in the number of sensors available to us, for the purpose of control of the model, we do not use the events which correspond to the operation of either of the loaders, or to that of the unloading conveyor. Instead we work with the information available to us, and assume the occurrence of events which imply other events. For example, the departure of the unloading pusher from its retracted position and its subsequent return imply that if a loaded fixture was present at the unloading section then that part has been offloaded. Note that although such situation can be handled by the SCT with partial observation [66], but that is beyond the scope of the present study.

- We assume that the parts are loaded only when the conveyers are stationary.

- We do not consider the case of motor or sensor failures.

- The controllable events in the system are assumed to be disabled unless explicitly enabled.

- We start our system assuming the transporter is referenced, i.e., $A5$ reads 0 rotations, all the motors are off, touch sensors indicating the chassis $T2$, roof $T3$ & unloading $L8$ light sensor pushers are retracted, i.e., "On", and that the roof press touch sensor $T4$ is raised. Refer to Figure 2.7.

2.5 Plant Models

The plant is modeled using deterministic FSM models. Instead of having a single incomprehensible model for the system, and to make FSM designing simpler, we develop smaller sized sub-models of the system. The entire system is partitioned into different sections and models are developed for each section separately. Thus the intricacies of having a large model is avoided, and the time of development required for the smaller models is substantially less.

The plant model is composed of individual models for the transporter, chassis, roof, press and unloading sections. Since it is the transporter that co-ordinates the entire assembly process, the primary model of the system as a whole is developed around it. It is implicit in all the models that those events that do not appear in the FSM of the particular section, appear as self-loops on all the states. *Self-loops* are just a way of indicating that the current state of this particular section is in no way affected by the occurrence of that event in some other section. To obtain the overall behavior of the plant, a strict synchronous operation of the different subsections is taken.

The actuator & sensor signals and their events used for modeling the plant are given in Figure 6.1. Each state in the plant model can potentially be one of the final states, which indicates completion of specific tasks. This fact is shown in the plant FSM models by using filled circles to mark the states. In order to determine which of the states is a final state, the progress specifications, described in Section 2.7, are used, as shown later in Section 2.8. Progress specifications determine the markings on the plant model.

2.5.1 Transporter (Fixture Slide)

This forms the backbone of the entire assembly line. Parts are transported from one assembly section to another via the transport mechanism, which essentially consists of a fixture that is connected to one end of a rack that is moved by a pinion powered from a gear box motor, **M1**. An angle sensor, **A5**, mounted on the same shaft as that of the pinion, counts off the number of rotations of the axle through it, in order to determine the

Section	Signals	Events	Controllable
Transporter	M1	Ifon/of: indexing slide motor on/off forward dirn. input events	Yes
	M1	Iron/of: indexing slide motor on/off reverse dirn. input events	Yes
	A5	a : Indexing slide leaving home station during forward movement output event	No
	A5	b/j : Indexing slide at press station during forward/reverse movement output events	No
	A5	c/k : Indexing slide leaving press station during forward/reverse movement o/p events	No
	A5	d/h : Indexing slide at roof station during forward/backward movement o/p events	No
	A5	g : Indexing slide leaving roof station during forward/reverse movement o/p event	No
	A5	f : Indexing slide at chassis station during reverse movement optput event	No
	A5	l : Indexing slide at home or unloading position during reverse movement o/p event	No
Chassis	M2	cCon/of: chassis conveyor motor on/off input events	Yes
	M3	pCon/of: chassis pusher motor on/off input events	Yes
	T2	pCup/dn: chassis pusher retracted/not−retracted output events	No
	L6	dCup/dn: part present/absent at chassis station dock output events	No
Roof	M4	cRon/of: roof conveyor motor on/off input events	Yes
	M5	pRon/of: roof pusher motor on/off input events	Yes
	T3	pRup/dn: roof pusher retracted/not−retracted output events	No
	L7	dRdn/up: part present/absent at roof station dock output events	No
Press	M6	pPfon/of: press pusher motor on/off input events	Yes
	M6	pPron/of: press pusher motor on/off in reverse dirn. input events	Yes
	M7	wPon/of: press winding motor on/off input events	Yes
	T1	pPup/dn: press pusher retracted/not−retracted output events	No
	T4	wPup/dn: press weight raised/lowered output events	No
Unloading	M8	pcUon/of: unloading pusher and conveyor motor on/off input events	Yes
	L8	pUup/dn: unloading pusher retracted/not−retracted output events	No

Additional legends used in FSM models

Σ : set of all events

Σ_c : set of all controllable events {Ifon/of, Iron/of, cCon/of, cPon/of, rCon/of, rPon/of, pPfon/of, Pron/of, wPon/of, pcUon/of}

Σ_{uc} : set of all uncontrollable events = $\Sigma - \Sigma_c$

Σ_I : set of all events in the transporter

$\Sigma_{(c)I}$ = { Ifon, Ifof, Iron, Irof}
Corresponding to transporter forward/reverse on/off

$\Sigma_{(uc)I}$ = { a, b, c, d, e, f, g, h, i, j, k, l }
Corresponding to different angle positions

ICdn : $\Sigma_{(uc)I}$ − { f } (transporter not at the chassis section)

IRdn : $\Sigma_{(uc)I}$ − { d, h }(transporter not at the roof section)

IPdn : $\Sigma_{(uc)I}$ − { b, j } (transporter not at the press section)

IUdn : $\Sigma_{(uc)I}$ − { l } (transporter not at unloading section)

Figure 2.8: Legend of signal and event labels

position of the fixture. The FSM model of the transporter is given in Figure 6.2. Initially the

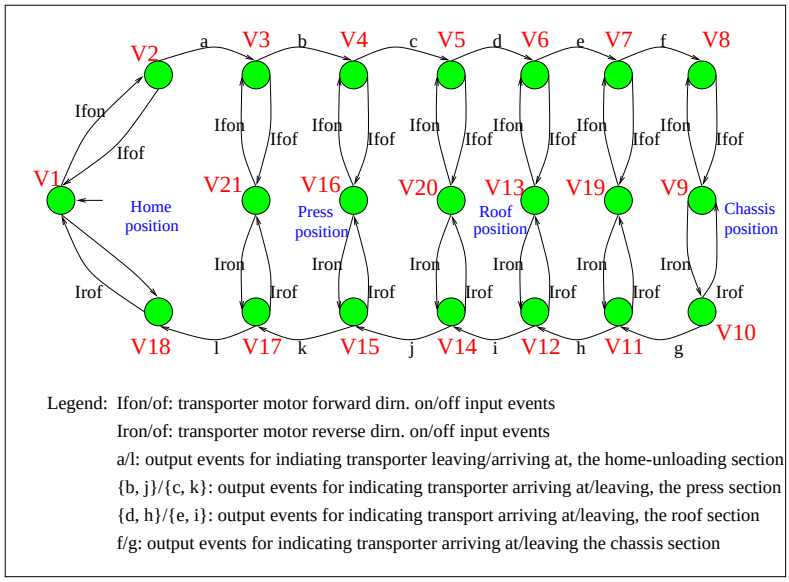

Figure 2.9: Transporter FSM model

transporter in the home position *V1* (refer Figure 6.2). It can either be switched on in the forward or reverse directions by the action of motor events *Ifon* and *Iron* respectively. Since from the home position only forward movement is possible, the motor event *Ifon* results in sensor event *a* indicating that the slide has left the home position. Proceeding in this fashion we model the movement of the slide until it reaches the chassis section. This is denoted by the sensor event *f*. At this state (*V8*), the forward movement can be switched off with the motor event (*Ifof*) bringing the system to state *V9*. Other states in the transporter FSM can be obtained by tracing out the actuator-sensor event flow for this section.

2.5.2 Chassis

The chassis conveyor conveys parts to its docking area. The chassis dock acts as a buffer with a capacity of one part. Parts are pushed off the dock by a chassis pusher, onto the empty fixture attached to the transporter. Sensors monitor the retracted position of the pusher and presence of part on the dock. The FSM models of the chassis section operations is given in Figure 2.10. It consists of two automata models. A 15 state automata shows the behavior of the plant events in the chassis section. An additional 2 state automata is shown to indicate that, owing to mechanical considerations, it is not possible for the chassis pusher

to advance unless the transporter slide with the fixture is stationed in front of the chassis station's loading dock.

2.5.3 Roof

The roof conveyor conveys parts that are loaded on it, onto the roof dock which also has a buffer size of one. A part is pushed off the dock onto an waiting transporter by the roof pusher. Sensors monitor the retracted position of the pusher and presence of part on the dock. The FSM model of the roof section operations is given in Figure 2.10.

2.5.4 Press

The pressing of the roof and the chassis is done by releasing a weighted LEGO block onto a properly positioned transporter carrying the roof-chassis combination. The mechanism is controlled by a press pusher and Winding motor. Initially the pusher is advanced so that the weighted block is suspended at a certain height. When the pusher retracts the weight descends and presses the pieces together. After this the pusher is advanced again so as to mesh with the winding motor gears, which when switched on raises the block up again. The retracted position of the pusher and the raised position of the block are monitored by sensors. The FSM model of the press section operations is given in Figure 2.11.

2.5.5 Unloading

The unloading conveyor conveys parts that are pushed onto it by the unloading pusher. There is a sensor for monitoring the retracted position of the pusher. The FSM model of the unloading section is given in Figure 2.11. consists of two automata. The behavior of the unloading section is modeled by the four state automata. Owing to mechanical considerations it is not possible for the unloading pusher to advance unless the transporter slide with the fixture is stationed in front of the unloading station's loading dock. This is indicated by the 2 state automata in Figure 2.11.

2.6 Safety Specification Models

Since the safe operation of the plant is mainly concerned with the safe enabling of controllable events, many of the models are drawn considering a particular output being on or

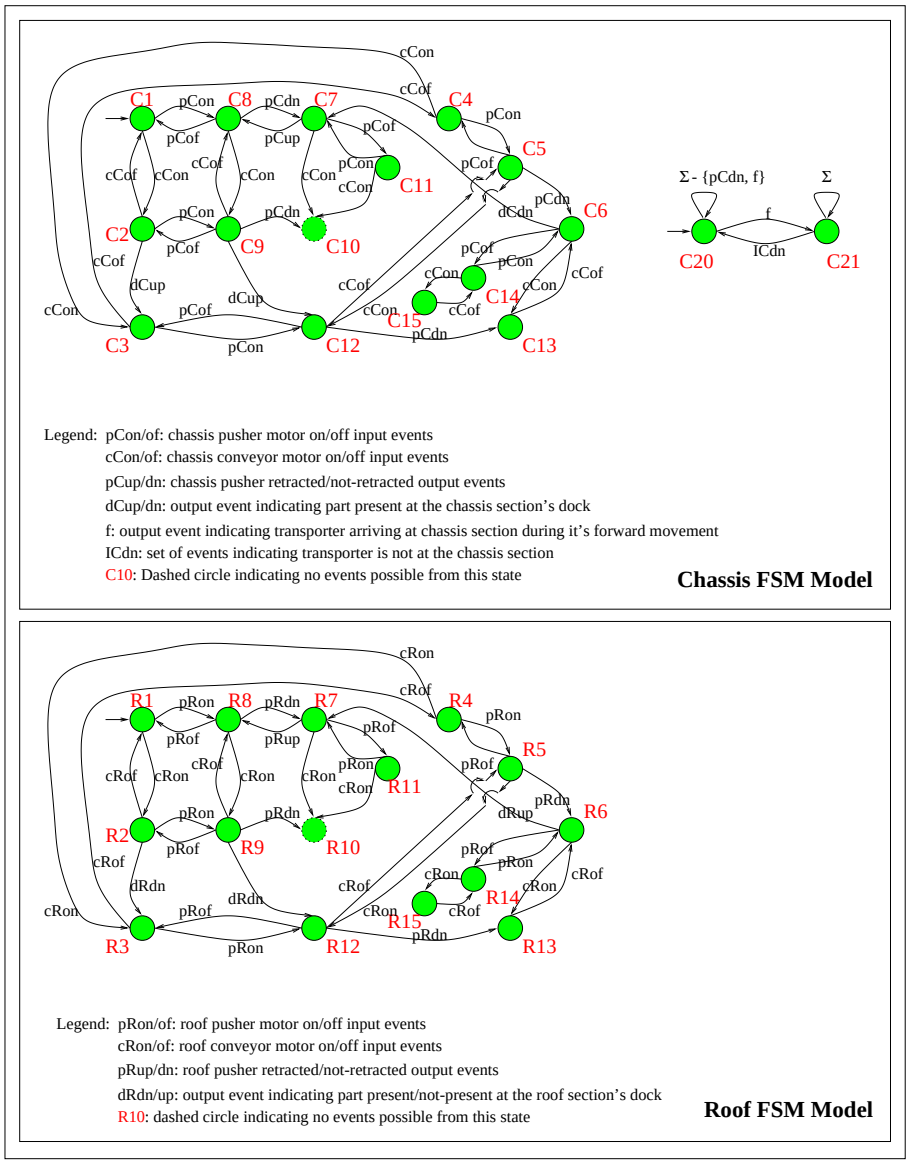

Figure 2.10: FSM models of the chassis and roof sections

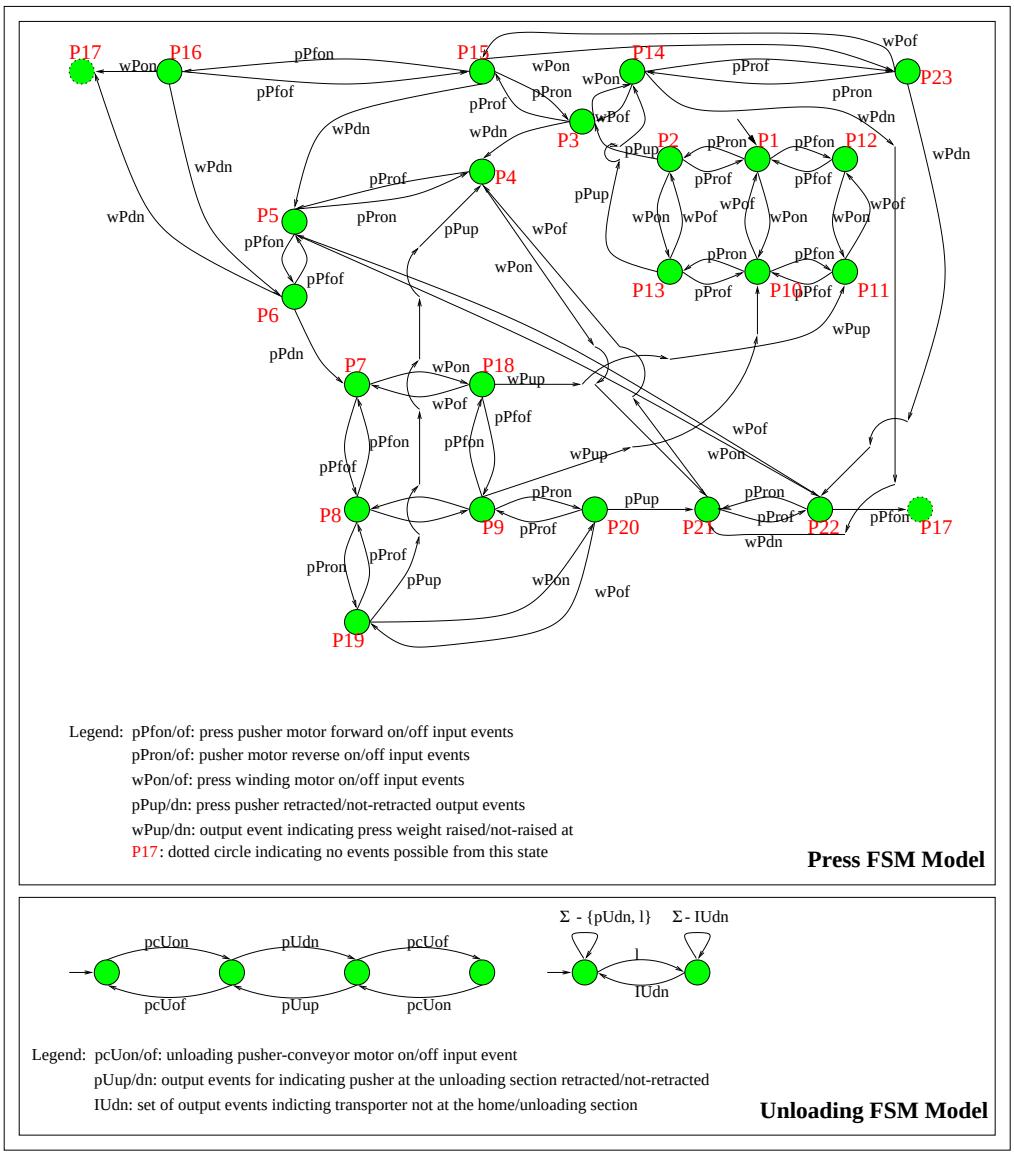

Figure 2.11: FSM models of the press and unloading sections

off, and whether some other transition can be safely turned on under that condition. There are a total of sixteen safety specifications for the entire plant model. Six of them are related to multiple sections, and hence are called global specifications. The remaining ten are related to the safety required for the operation of individual sections, and hence are called local.

2.6.1 Global Safety Specifications

Many of the specifications require interactions between the transporter and the different sections of the assembly line. For the purpose of modeling it is convenient to group many of the uncontrollable transporter events listed in Figure 2.12.

The 6 global safety specifications for the plant are enumerated next and their FSM models are given in Figure 2.12:

1. **Global safety specification 1,** $K1$ (Figure 2.12)
   This model pertains to the chassis section and to the transporter. The chassis pusher **M3** should not advance if the transporter is not at the chassis section i.e. **A5** $= 80$.

2. **Global safety specification 2,** $K2$ (Figure 2.12)
   This model pertains to the roof section and to the transporter. The roof pusher **M5** should not advance if the transporter is not at the roof section i.e. **A5** $= 52$.

3. **Global safety specification 3,** $K3$ (Figure 2.12)
   This model pertains to the press section and to the transporter. The transporter **M1** should not move if the press is lowered i.e. **T4** off.

4. **Global Safety specification 4,** $K4$ (Figure 2.12)
   This model pertains to the unloading section and to the transporter. The unloading pusher **M8** should not advance if the transporter is not in the home/unloading position **A5** $= 0$.

5. **Global safety specification 5,** $K5$ (Figure 2.12)
   This model pertains to the press section and to the transporter. The press pusher **M6** should not move back if the transporter is not under the press section **A5** $= 21$.

6. **Global safety specification 6,** $K6a, K6b$ (Figure 2.12)
   This model pertains to the press section, the unloading section and to the transporter.

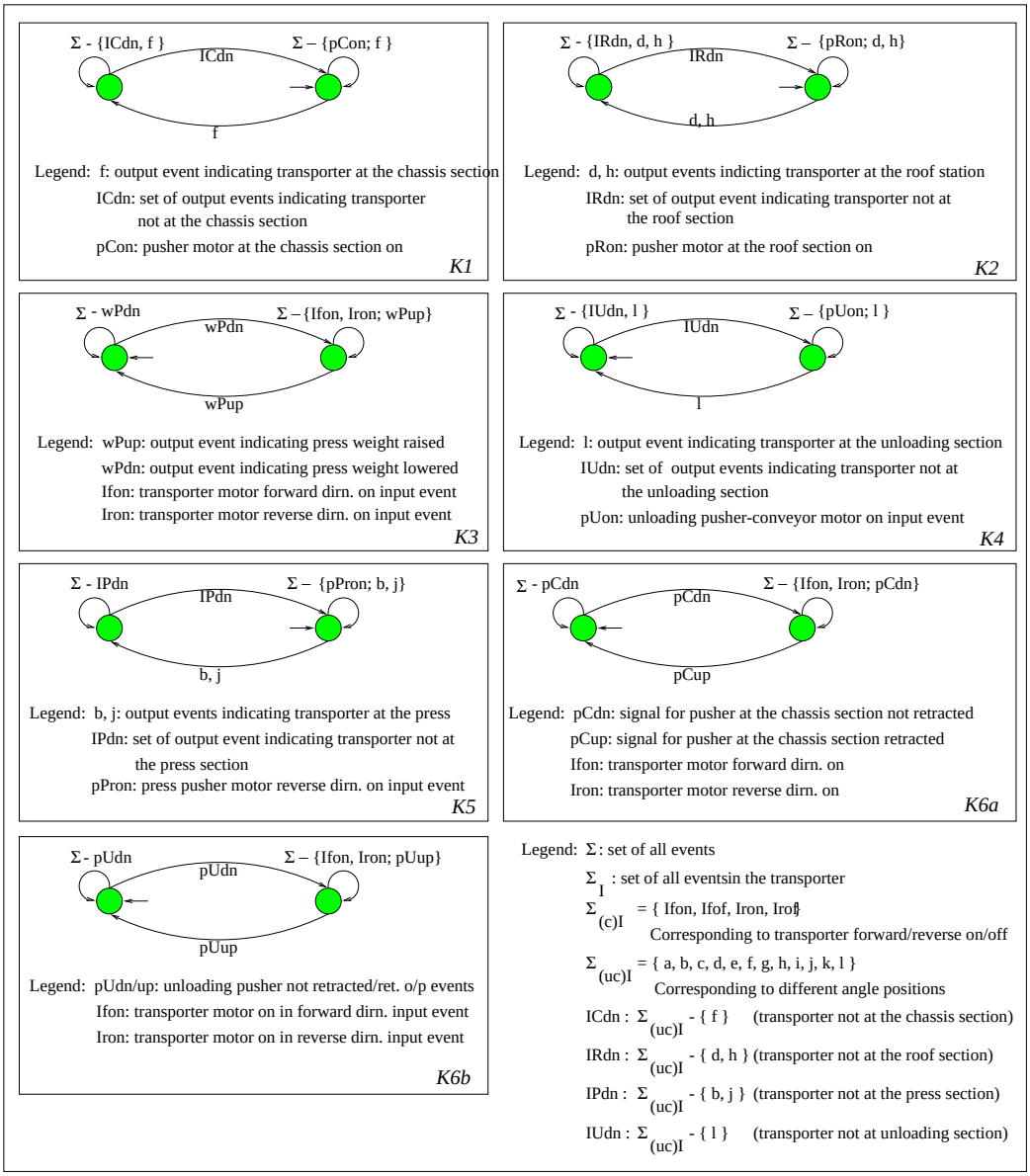

Figure 2.12: Global safety specifications, $K1 - K6$

The transporter **M1** should not move unless the retracted signals from the chassis pusher **T2** and the unload pusher **L8** are received.

2.6.2 Local Safety Specifications

There are ten such specifications, 3 for the chassis and the roof each, and 2 for the press and the transporter each. The FSM models for these specifications are given in Figure 2.13.

1. **Local safety specification 1,** $K7$ (Figure 2.13)
   This model pertains to the chassis section. If the chassis pusher **T2** is not retracted, the chassis conveyor **M2** cannot be switched on, and the chassis pusher **M3** be prevented from turning off.

2. **Local safety specification 2,** $K8$ (Figure 2.13)
   This model pertains to the chassis section. If the chassis pusher motor **M3** is on, the chassis conveyor **M2** may not be switched on and vice versa.

3. **Local safety specification 3,** $K9$ (Figure 2.13)
   This model pertains to the chassis section. If there is a part on the chassis dock **L6**, then the chassis conveyor **M2** should be switched off. On the other hand when there is no part on the dock then the chassis pusher **M3** should not be switched on.

4. **Local safety specification 4,** $K10$ (Figure 2.13)
   This model pertains to the roof section. If the roof pusher **T3** is not retracted, the chassis conveyor **M4** cannot be switched on, and the roof pusher **M5** be prevented from turning off.

5. **Local safety specification 5,** $K11$ (Figure 2.13)
   This model pertains to the roof section. If the roof pusher motor **M5** is on, the roof conveyor **M4** may not be switched on and vice versa.

6. **Local safety specification 6,** $K12$ (Figure 2.13)
   This model pertains to the roof section. If there is a part on the roof dock **L7**, then the roof conveyor **M4** should be switched off. On the other hand when there is no part on the dock then the chassis pusher **M5** should not be switched on.

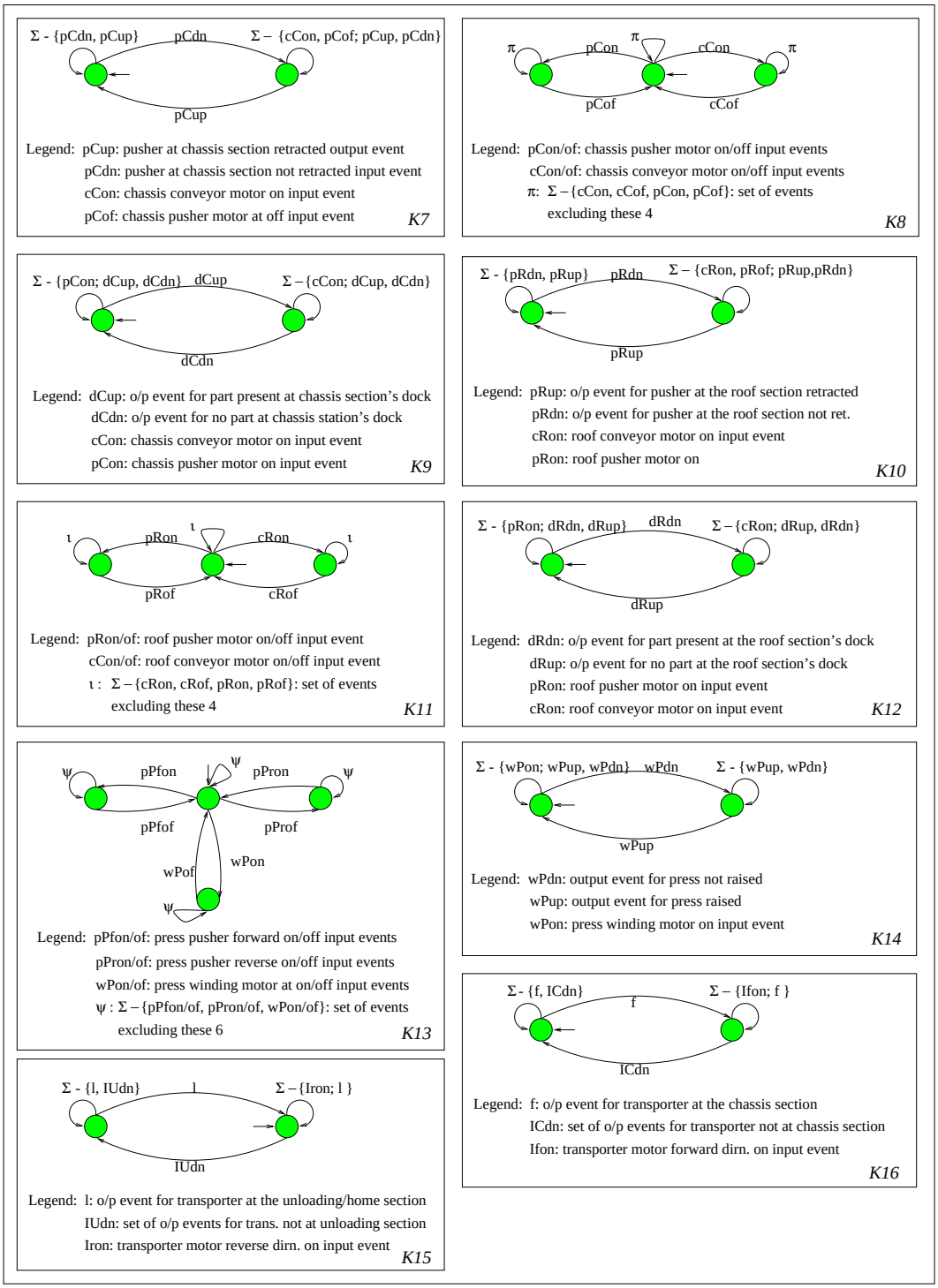

Figure 2.13: Local safety specifications, $K7 - K16$

7. **Local safety specification 7,** $K13$ (Figure 2.13)
   This model pertains to the press section. If the press pusher motor **M6** is on in either direction, the press winding motor **M7** may not be switched on and vice versa.

8. **Local safety specification 8,** $K14$ (Figure 2.13)
   This model pertains to the press section. The winding motor **M7** should be stopped if the press up signal **T4** exists.

9. **Local safety specification 9,** $K15$ (Figure 2.13)
   Backward motion of the referenced transporter **M1** is not permitted if the angle sensor reads **A5** 0 rotations, i.e., when it is in its initial state. This pertains to the transporter.

10. **Local safety specification 10,** $K16$ (Figure 2.13)
    This model pertains to the transporter. Forward movement of of the referenced transporter **M1** is not permitted if the angle sensor reads **A5** 80 rotations, i.e., it is at the chassis section.

2.7 Progress Specification Models

The progress specification is specific to a particular product being assembled, and essentially requires a certain order of uncontrollable events to be followed. For the present example this governs the way in which the assembly of the automobile is done in the plant. It is represented by the specification language $K17$.

The overall task is broken down into the cyclic execution of the four subtasks:

- Move the transporter to the chassis section and start chassis operations (sub-task ST1).
- Move the transporter to the roof section and start roof operations (sub-task ST2).
- Move the transporter to the press section and start roof operations (sub-task ST3).
- Move the transporter to the unloading section and start unloading operations (sub-task ST4).

Refer to Figure 2.14 for the procedure followed in the assembly line under this progress specification.

The progress specification has only one state marked, the initial one, indicating that the final and initial positions are the same and that the cycle can repeat indefinitely.

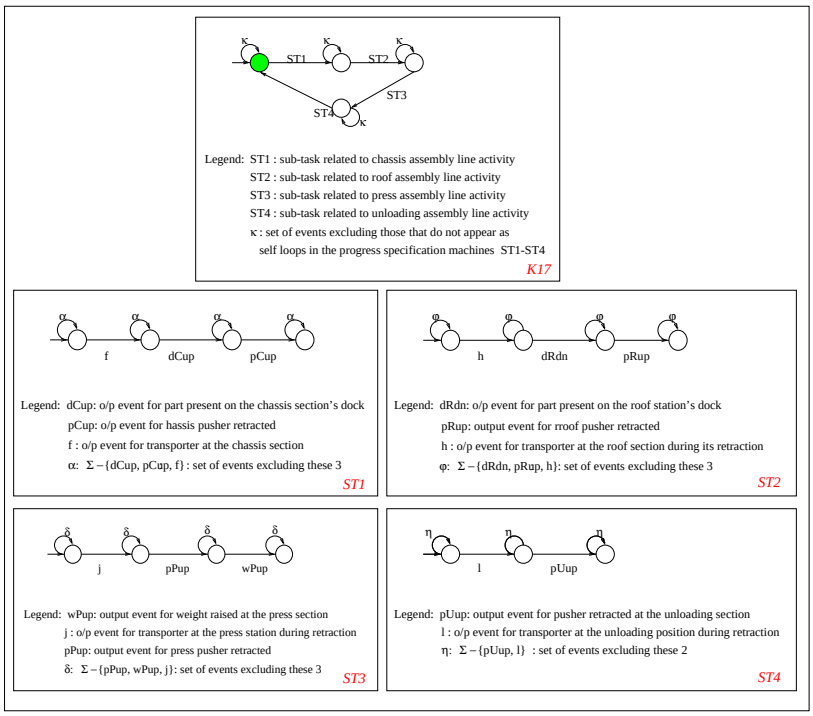

Figure 2.14: Global progress specification, $K17$, and sub-tasks $ST1 - ST4$

Next we present the individual sub-task specification models.

1. **Sub-Task 1**

   Refer to Figure 2.14 for the progress specifications which is related to the chassis section wherein the chassis conveyor starts causing a part to be delivered to the chassis dock. The chassis pusher then comes on, delivering the part to the fixture attached to the transporter. The chassis section needs to perform the following sequence of operations:

   - Wait for the transporter to reach the chassis section (f).
   - Wait until light sensor **L6** turns on ($dCup$), indicating part on dock.
   - Wait until when sensor **T2** turns on ($pCup$), indicating chassis pusher back.

2. **Sub-Task 2**

   Refer to Figures 2.14 for the progress specifications which is related to the roof section wherein the roof conveyor starts causing a part to be delivered to the roof dock. The roof pusher then comes on delivering the part to the waiting transporter's fixture. The roof section needs to perform the following sequence of operations:

- Wait for the transporter to reach the roof section (h).
- Wait until light sensor **L7** turns off ($dRdn$), indicating part on dock.
- Wait until when sensor **T3** turns on ($pRup$), indicating roof pusher back.

3. **Sub-Task 3**
   Refer to Figures 2.14, for the progress specifications which is related to the press section wherein the press pusher retracts causing the press to descend. Immediately after that the press pusher advances, and engages with the winding motor. The winding motor stars up and the press is raised the top position being sensed by a sensor. The press section needs to perform the following sequence of operations:

   - Wait for the transporter to reach the press section (j).
   - Wait for sensor **T1** to be turn on ($pPup$).
   - Wait for **T4** to turn on, indicating that the press is raised ($wPup$).

4. **Sub-Task 4**
   Refer to Figure 2.14 for the progress specifications which is related to the unloading section wherein the unloading conveyor starts causing a part delivered to be the staging area. The unloading section needs to perform the following sequence of operations:

   - Wait for the transporter to reach the unloading section (l).
   - Wait until light sensor **L8** turn on ($pUup$), indicating part on dock.

Remark 1 The progress specifications may vary for different possible objectives for which the plant is being used (the desired flow of operations in different sections and for the system as a whole will vary). Thus, for example, instead of making an automobile, we may instead choose to do a simple transport activity. This would cause the progress specifications to be altered, even though the previous safety specifications will still needed to be enforced.

2.8 Supervisor Synthesis and Controller Extraction

For the purpose of supervisor synthesis, each of the plant sub-models is first completed with respect to the entire alphabet of the system. This is done by adding self-loops on those transitions that do not exist is a particular sub-machine, but are present in the alphabet

of the plant. Next a synchronous composition of these sub-models is taken yielding a state space for the plant models of size around 8×10^5. On proceeding in a similar fashion with the specification models we obtain their size to be that of comparable order as that of the plant. For computing a supervisor, we use the supervisory control toolkit available at the University of Kentucky (`www.engr.uky.edu/~kumar`). The overall safe and progress specification is determined to be controllable, and relative-closed and hence the overall specification FSM serves as the maximally permissive supervisor.

In order to manage the computational complexity, we use a modular approach to verify the controllability and relative-closure of the intersection of all safety ($K1 - K16$) and progress ($K17$) specifications. First we verify the controllability of individual safety specifications against the relevant portions of the plant. For example, the first local safety specification $K7$ concerns the operation of the chassis section. So we verify its controllability against the model of the chassis section only, which as shown in Figure 2.10 has 30 states. Since $K7$ has 2 states, the complexity of this verification is $O(60)$. Proceeding in a similar fashion the controllability of other local safety specifications is verified. Next, for each global safety specification which involves more than one section, we use the synchronous composition of the FSM models of the relevant sections as the plant model and reform the controllability test. For example, the plant model for the sixth global safety specification $K6$ comprises of the synchronous composition of the FSM models of the chassis, the transporter, and the unloading sections. Finally, since the individual safety specifications are controllable and prefix closed we conclude that their intersection is also controllable and prefix closed.

It remains to verify the controllability and relative-closure of the progress specification $K17$, and its non-conflictingness with respect to the intersection of all the safety specifications $K := \cap_{i=1}^{16} Ki$. Since the plant marking is determined solely by the progress specification, $L_m(G) := K17 \cap L(G)$. Then we have

$$\overline{K17} \cap L_m(G) = \overline{K17} \cap K17 \cap L(G) \subseteq K17,$$

i.e., $K17$ is relative-closed. We establish the controllability of $K17$ by viewing it as a cyclical concatenation of the four sub-tasks. The first sub-task starts in the initial state of the plant and upon completion sends the plant to a final state, which can be treated as the initial state of the second sub-task. We verify the controllability of each of the subtasks against the relevant portion of the plant which is appropriately initialized and terminated. This let us perform the controllability test of the progress specification modularly, where modularity stems from the sequential (as opposed to parallel) decomposition of the progress

specification.

Next we establish that the intersection of the safety specifications K and the progress specification $K17$ is non-conflicting. We observe that the progress specification never violates any of the safety, i.e., $\overline{K17} \subseteq K = \overline{K}$, which also implies $K17 \subseteq K = \overline{K}$. These automatically give us the non-conflictingness property since,

$$\overline{K17} \cap \overline{K} = \overline{K17} = \overline{K17 \cap K}.$$

Finally we are interested in obtaining a controller for the supervised plant. One particular controller candidate, having 49 states, is shown in Figure 2.15. It is obtained by selecting at most one controllable event, when possible, from among the ones allowed by the maximally permissive supervisor. Note that at any stage of the assembly process the controller enables at most one actuator (controllable) event to occur, and then waits for the response of the system in the form of a sensor (uncontrollable) event. This procedure is repeated until the task specified by the progress specification is completed while always following operating in the safe region.

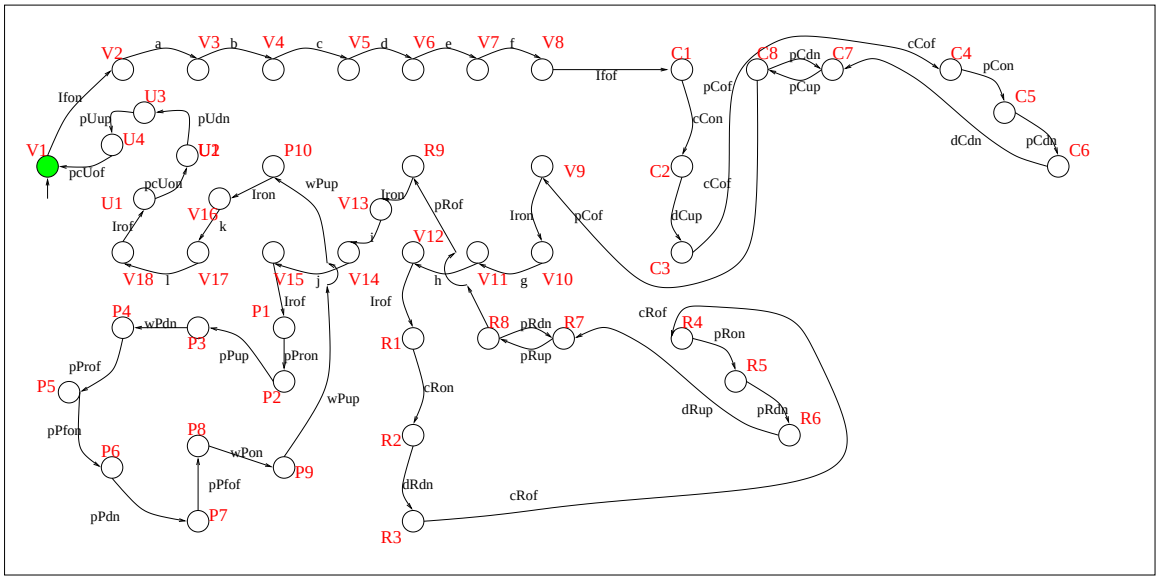

Figure 2.15: Controller for LEGO assembly line

The controller chosen for implementation is translated into LEGO commands in a direct way, as there exists a one-to-one correspondence between plant events and the actuator commands to be executed or the sensor inputs to be monitored. A portion of the control lab code corresponding to Figure 2.15, is provided as reference. Only in those places where

a controllable event needs to be enabled is the control lab code inserted. Those states in the FSM where only uncontrollable events occur and have no controllable transitions leading out of them need no additional action to be enabled by the controller. So, for example, when the transporter is moving from the home/unloading section to the chassis section it monitors the occurrence of events corresponding to its crossing the press and roof sections but does not take any other action than monitor them. This is illustrated in the following Lego control code fragment.

```
to main                    ; call the control program
  tto "M1                  ; reference M1, the transporter motor
  setright                 ; set direction of movement ("Ifon")
  on                       ; turn motor on
  waituntil [A5=80] off    ; monitor value of A5, the angle sensor at
                           ; until it reaches 80 ("f") at chassis,
                           ; section, then stop motor M1 ("Ifof")
  .
  .
  .

  launch[main]             ; restart the control program for
                           ; cyclic operation
end                        ; end of the control program
```

We choose a control scheme wherein for those states of the FSM where both a controllable and uncontrollable event are permitted to occur, and if the controllable event is such that its execution at this state will prohibit the occurrence of the uncontrollable event at any further state we choose to execute the controllable one, rather than wait for the uncontrollable one to occur. Such a case occurs in the Lego system while the transporter moves between different section and we are able to control its movement its position precisely by switching it off. Given the time constants in the system it is thus feasible to effectively pre-empt the occurrence of uncontrollable events in some states.

The software has capability of switching outputs on and off; waiting until a variable has reached certain boolean values prior to executing a command; certain amount of multitasking for launching sub-processes that will run in the background; and repetitions of a desired task sequences.

2.9 Conclusion

Implementation of the supervisory control theory for control of a miniature assembly line built from LEGO blocks, has been carried out. The controller so derived is by its construction guaranteed to be safe in operation, while also ensuring that the task for which it has been constructed will be accomplished.

Any change in the system such as addition or removal of equipment and the expansion of system operations to include new products types, can be easily incorporated by drawing new FSM models of the specific components of the plant and changing the relevant safety, progress specifications finite state machine models. Then, using supervisory control theory, a new supervisor can be synthesized, and a controller extracted from it. This approach is a generic one and can be applied to discrete event systems which are modeled as FSMs.

Chapter 3

Polynomial Time Diagnosis Algorithm for DESs

3.1 Introduction

Failure diagnosis is a critical task in large and complex systems. This problem has received considerable attention in the literature of various domains including the discrete event systems [14, 39, 67, 98, 93, 111]. In [98], Sampath *et al.* proposed a failure diagnosis approach for discrete event systems. They introduced the notion of diagnosability and gave a necessary and sufficient condition for testing it. Their condition is expressed as a property of the diagnoser of the system. In order to test the diagnosability, the diagnoser needs to be constructed first. The complexity of constructing the diagnoser and testing the diagnosability is exponential in the number of states of the system and doubly exponential in the number of failure types.

It is clear that if we could test more efficiently whether or not a system is diagnosable without having to construct a diagnoser, it would save us the time involved in constructing a diagnoser for the system which may not be diagnosable. In this chapter, we give a method for testing the diagnosability without having to construct a diagnoser. The complexity of our method is polynomial in the number of states of the system and also in the number of failure types.

In the rest of the chapter, we first introduce the notion of diagnosability of discrete event systems, then present our algorithm for testing it. Finally, an illustrative example is provided.

3.2 Diagnosability

We first give the system model and then define the diagnosability as introduced by [98].

3.2.1 System model

Let $G = (X, \Sigma, \delta, x_0)$ be a finite state machine model of the system to be diagnosed, where

- X is a finite set of states;
- Σ is a finite set of event labels;
- $\delta \subseteq X \times \Sigma \times X$ is a finite set of transitions;
- $x_0 \in X$ is the initial state.

We assume that all state machines are accessible (all states can be reached from the initial state), and otherwise we consider only the accessible part of the state machine. We let Σ^* denote the set of all finite length event sequences, including the zero length sequence denoted ϵ. An element of Σ^* is called a trace, and a subset of Σ^* is called a language. For a trace s and an event σ, we write $\sigma \in s$ to imply that σ is an event contained in the trace s. A path in G is a sequence of transitions $(x_1, \sigma_1, x_2, \ldots, \sigma_n, x_n)$ such that for each $i \in \{1, \ldots, n-1\}$, $(x_i, \sigma_i, x_{i+1}) \in \delta$; this path is a cycle if $x_n = x_1$. We use $L(G) \subseteq \Sigma^*$ to denote the generated language of G, i.e., the set of traces that can be executed in G starting from its initial state. Then $L(G)$ is prefix-closed, i.e., $L(G) = pr(L(G))$, where $pr(L(G)) = \{u | \exists v \in \Sigma^*, uv \in L(G)\}$ is the set of prefixes of traces in $L(G)$. Let $\Sigma_o \subseteq \Sigma$ denote the set of observable events, $\Sigma_{uo} = \Sigma - \Sigma_o$ be the set of unobservable events, $M : \Sigma \to \Sigma_o \cup \{\epsilon\}$ be the observation mask, $\mathcal{F} = \{F_i, i = 1, 2, \ldots, m\}$ be the set of failure types, $\psi : \Sigma \to \mathcal{F} \cup \{\emptyset\}$ be the failure assignment function for each event in Σ. The definition of M is extended from Σ to Σ^* inductively as follows: $M(\epsilon) = \epsilon$ and for each $s \in \Sigma^*, \sigma \in \Sigma : M(s\sigma) = M(s)M(\sigma)$.

We make the following assumptions as in [98] for the system studied in this chapter.

A1 The language $L(G)$ generated by G is live. This means that there is a transition defined at each state x in X.

A2 There does not exist in G any cycle of unobservable events, i.e., $(\exists k \in N)\ (\forall ust \in L(G), s \in \Sigma_{uo}^*) \Rightarrow ||s|| \leq k$, where N denotes the set of natural numbers, and $||s||$ denotes the length of trace s.

A3 Every failure event is unobservable, i.e., $(\forall \sigma \in \Sigma,\ \psi(\sigma) \neq \emptyset) \Rightarrow M(\sigma) = \epsilon$.

3.2.2 Diagnosability

The diagnosability for discrete event systems defined in [98] is described as follows:

Definition 1 A prefix-closed language L is said to be diagnosable with respect to the observation mask M and the failure assignment function ψ if the following holds:

$(\forall F_i \in \mathcal{F})\ (\exists n_i \in N)\ (\forall s \in L, \psi(s_f) = F_i)\ (\forall v = st \in L, ||t|| \geq n_i)$

$\Rightarrow (\forall w \in L, M(w) = M(v))\ (\exists u \in pr(\{w\}), \psi(u_f) = F_i)$,

where s_f and u_f denote the last events in traces s and u respectively, $pr(\{w\})$ is the set of all prefixes of w. A system G is said to be diagnosable if its language $L(G)$ is diagnosable.

The above definition states that if s is a trace in L ending with a F_i-type failure, and v is a sufficient long (at least n_i events longer) trace obtained by extending s in L, then every trace w in L that is observation equivalent to v, i.e., $M(w) = M(v)$, should contain in it a F_i-type failure.

3.3 Algorithm

We now present the algorithm for testing the diagnosability.

Algorithm 1 For a given system $G = (X, \Sigma, \delta, x_0)$ with an observation mask M and a failure assignment function ψ, do the following:

1. Obtain a nondeterministic finite state machine $G_o = (X_o, \Sigma_o, \delta_o, x_0^o)$ with language $L(G_o) = M(L(G))$ as follows:

   - $X_o = \{(x, f) \mid x \in X_1 \cup \{x_0\}, f \subseteq \mathcal{F}\}$ is the finite set of states, where $X_1 = \{x \in X \mid \exists (x', \sigma, x) \in \delta$ with $M(\sigma) \neq \epsilon\}$ is the set of states in G that can be reached through an observable transition, and f is the set of failure types along certain paths from x_0 to x.
   - Σ_o, the set of observable events, is the set of event labels for G_o.
   - $\delta_o \subseteq X_o \times \Sigma_o \times X_o$ is the set of transitions. $((x, f), \sigma, (x', f')) \in \delta_o$ if and only if there exists a path $(x, \sigma_1, x_1, \ldots, \sigma_n, x_n, \sigma, x')$ $(n \geq 0)$ in G such that $\forall i \in \{1, 2, \ldots, n\}, M(\sigma_i) = \epsilon$, $M(\sigma) = \sigma$, and $f' = \{\psi(\sigma_i) \mid \psi(\sigma_i) \neq \emptyset, 1 \leq i \leq n\} \cup f$.

- $x_0^o = (x_0, \emptyset) \in X_o$ is the initial state.

2. Compute $G_d = (G_o||G_o)$, the strict composition of G_o with itself. Then $G_d = (X_d, \Sigma_o, \delta_d, x_0^d)$, where

   - $X_d = \{(x_1^o, x_2^o) \mid x_1^o, x_2^o \in X_o\}$ is the set of states.
   - Σ_o is the set of event labels for G_d.
   - $\delta_d \subseteq X_d \times \Sigma_o \times X_d$ is the set of transitions. $((x_1^o, x_2^o), \sigma, (y_1^o, y_2^o)) \in \delta_d$ if and only if both (x_1^o, σ, y_1^o) and (x_2^o, σ, y_2^o) are in δ_o.
   - $x_0^d = (x_0^o, x_0^o) \in X_d$ is the initial state.

3. Check whether there exists in G_d a cycle $cl = (x_1, \sigma_1, x_2, \ldots, x_n, \sigma_n, x_1)$, $n \geq 1$, $x_i = ((x_i^1, f_i^1), (x_i^2, f_i^2))$, $i = 1, 2, \ldots, n$, such that $f_1^1 \neq f_1^2$. If the answer is yes, then output that the system is not diagnosable; otherwise output that the system is diagnosable. This last step can be performed by first identifying states $((x^1, f^1), (x^2, f^2))$ in G_d for which $f^1 \neq f^2$, and deleting all the other states and the associated transitions; and next checking if the remainder graph contains a cycle.

In the following, we give two Lemmas showing some properties of the state machines G_o and G_d derived in Algorithm 1. The proofs are omitted here because they follow directly from the definitions of G_o and G_d.

Lemma 1 For the state machine G_o the following holds:

1. $L(G_o) = M(L(G))$.

2. For every path tr in G_o ending with a cycle,

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

   we have

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

Lemma 2 For every path tr in G_d ending with a cycle,

$$tr = (x_0^d, \sigma_0, x_1, \ldots, x_k, \sigma_k, \ldots, x_n, \sigma_n, x_k),$$

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

1. there exist two paths tr_1 and tr_2 in G_o ending with cycles, namely,

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

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

Next we provide a theorem which guarantees the correctness of Algorithm 1.

Theorem 1 G is diagnosable if and only if for every cycle cl in G_d,

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

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

Proof: For the necessity, suppose G is diagnosable, but there exists a cycle cl in G_d, $cl = (x_k, \sigma_k, x_{k+1}, \ldots, x_n, \sigma_n, x_k)$, $n \geq k$, $x_i = ((x_i^1, f^1), (x_i^2, f^2))$, $i = k, k+1, \ldots, n$, such that $f^1 \neq f^2$. Since G_d is accessible, there exists a path tr in G_d ending with the cycle cl, i.e., $tr = (x_0^d, \sigma_0, x_1, \ldots, x_k, \sigma_k, \ldots, x_n, \sigma_n, x_k)$. Then from Lemma 2 we know that there exist two paths tr_1 and tr_2 in G_o with

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

Further from Lemma 1, we have $\exists u_1 v_1^*, u_2 v_2^* \in L(G)$ such that $M(u_1) = M(u_2) = \sigma_0 \ldots \sigma_{k-1}$, $M(v_1) = M(v_2) = \sigma_k \ldots \sigma_n$, and $\{\psi(\sigma) \mid \sigma \in u_i, \psi(\sigma) \neq \emptyset\} = \{\psi(\sigma) \mid \sigma \in u_i v_i, \psi(\sigma) \neq \emptyset\} = f^i$, $i = 1, 2$. Since $f^1 \neq f^2$, we suppose $F_k \in f^1 - f^2 \neq \emptyset$. Then $\exists s \in L(G)$ such that $\psi(s_f) = F_k$ and $u_1 = st$ for some $t \in \Sigma^*$. For any integer n_k, we can choose another integer ℓ such that $||tv_1^\ell|| > n_k$. Now we have $M(u_2 v_2^\ell) = M(stv_1^\ell)$ and $\{\psi(\sigma) \mid \sigma \in u_2 v_2, \psi(\sigma) \neq \emptyset\} = f^2$, which means that no failure event of type F_k is contained in $u_2 v_2^\ell$. So from the definition of diagnosability, G is not diagnosable. A contradiction to the hypothesis. So the necessity holds.

For the sufficiency, suppose for every cycle cl in G_d, $cl = (x_1, \sigma_1, x_2, \ldots, x_n, \sigma_n, x_1)$, $n \geq 1$, $x_i = ((x_i^1, f^1), (x_i^2, f^2))$, $i = 1, 2, \ldots, n$, we have $f^1 = f^2$. From the second clause of Lemma 2, we know that the hypothesis implies that $\forall x = ((x^1, f^1), (x^2, f^2)) \in X_d$, if $f^1 \neq f^2$ then x is not contained in a loop. It further implies that for any state sequence $(x_1, x_2, \cdots, x_k)$ in G_d with $x_i = (x_i^1, f_i^1), (x_i^2, f_i^2))$ for $1 \leq i \leq k$, if $f_i^1 \neq f_i^2$ for all $i \in \{1, 2, \cdots, k\}$, then the length of the state sequence is bounded by the number of states in G_d, i.e., $k \leq |X_d|$.

Now let s be a trace in $L(G)$ ending with a F_k-type failure event, i.e., $\psi(s_f) = F_k$, we claim that $\forall v = st \in L(G)$ with $||t|| > |X_d| \times (|X| - 1)$, $\forall w \in L(G)$ with $M(w) = M(v)$, there is a F_k-type failure event contained in w. From above, for any state $x \in X_d$ that can be reached from x_0^d by executing $M(s)$ in G_d, we have that for any state sequence starting from x in G_d, a state $y = ((y^1, f^1), (y^2, f^2)) \in X_d$ with $f^1 = f^2$ can be reached within $|X_d| - 1$ steps. This implies that $\forall v = st \in L(G)$ with $||M(t)|| > |X_d| - 1$, $\forall w \in L(G)$ with $M(w) = M(v)$, there is a F_k-type failure event contained in w. Further from the assumption that no unobservable cycle exists in G, each "observed event" in $M(t)$ can be preceded/followed by at most $|X|-1$ unobserved events. It follows that for the trace t above, $||t|| \leq (||M(t)|| + 1) \times (|X| - 1)$, i.e., $||M(t)|| \geq \frac{||t||}{|X|-1} - 1$. So if $||t|| > |X_d| \times (|X| - 1)$, then $||M(t)|| \geq \frac{||t||}{|X|-1} - 1 > \frac{|X_d| \times (|X|-1)}{|X|-1} - 1 = |X_d| - 1$, establishing our claim. (Note that we have assumed implicitly that $|X| > 1$; otherwise if $|X| = 1$, then from the assumption of no unobservable loops, no transition labeled by a failure event exists, so that the system is trivially diagnosable.) It follows from Definition 1 that G is diagnosable. So the sufficiency also holds. ■

Remark 2 From Algorithm 1, we know that the number of states in G_o is at most $|X| \times 2^{|\mathcal{F}|}$, the number of transitions in G_o is at most $|X|^2 \times 2^{2|\mathcal{F}|} \times |\Sigma_o|$. Since $G_d = G_o || G_o$, the number of states in G_d is at most $|X|^2 \times 2^{2|\mathcal{F}|}$, and the number of transitions in G_d is at most $|X|^4 \times 2^{4|\mathcal{F}|} \times |\Sigma_o|$.

The complexity of performing step 1 of Algorithm 1, which construct G_o, is thus $O(|X|^2 \times 2^{2|\mathcal{F}|} \times |\Sigma_o|)$, whereas that of step 2 of Algorithm 1, which construct G_d, is thus $O(|X|^4 \times 2^{4|\mathcal{F}|} \times |\Sigma_o|)$. The complexity of performing step 3 of Algorithm 1, which detects the presence of a certain "offending" cycle in an appropriately pruned subgraph of G_d (see the last sentence of step 3 of Algorithm 1), is linear in the number of states and transitions of the subgraph, i.e., it is $O(|X|^4 \times 2^{4|\mathcal{F}|})$. Note that while detecting the presence of a "offending" cycle, the transition labels are irrelevant.

So the complexity of Algorithm 1 is $O(|X|^4 \times 2^{4|\mathcal{F}|} \times |\Sigma_o|)$ which is polynomial in the

number of states in G and exponential in the number of failure types in G.

In [98], another necessary and sufficient condition was given for diagnosability. The condition was expressed as a property of a certain diagnoser of the system. So in order to check the diagnosability we needed to first construct the diagnoser, then check the property on the diagnoser. The complexity to construct the diagnoser as well as the complexity to check the property on the diagnoser is exponential in the number of states of the system and doubly exponential in the number of failure types of the system. In Algorithm 1, no diagnoser is needed for checking the diagnosability.

Remark 3 The complexity of testing diagnosability can be made polynomial in the number of fault types as well by noting that a system is diagnosable with respect to the fault types $\mathcal{F} = \{F_i, i = 1, 2, \cdots, m\}$ if and only if it is diagnosable with respect to the each individual fault type $F_i, i = 1, 2, \cdots, m$. In other words, one can apply Algorithm 1 m different times for testing diagnosability with respect the individual failure type sets $\{F_1\}, \cdots, \{F_m\}$. Since now each failure type set is a singleton, from Remark 2 it follows that the complexity of each such test is $O(|X|^4 \times 2^{4|1|} \times |\Sigma_o|) = O(|X|^4 \times |\Sigma_o|)$. So, the overall complexity of testing diagnosability is $O(|X|^4 \times |\Sigma_o| \times |\mathcal{F}|)$.

Example 1 Consider the system $G = (X, \Sigma, \delta, x_0)$:

- $X = \{x_0, x_1, x_2, x_3, x_4\}$
- $\Sigma = \{\sigma_1, \sigma_2, \sigma_3, \sigma_{uo}, \sigma_{f1}, \sigma_{f2}, \sigma_{f3}\}$
- $\delta = \{(x_0, \sigma_1, x_1), (x_1, \sigma_{f1}, x_2), (x_1, \sigma_{uo}, x_2), (x_2, \sigma_{f2}, x_3),$
  $(x_3, \sigma_2, x_3), (x_2, \sigma_{f1}, x_4), (x_4, \sigma_3, x_4)\}$

with the observable event set $\Sigma_o = \{\sigma_1, \sigma_2, \sigma_3\}$. The system is shown in Figure 3.1. Let

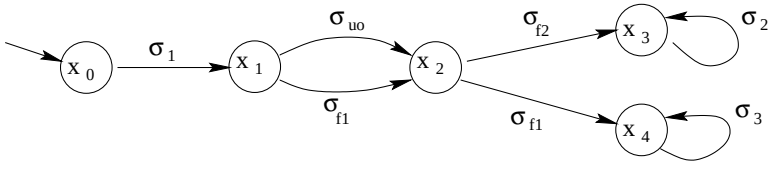

Figure 3.1: Diagram of the system G

$\mathcal{F} = \{F_1, F_2\}$ be the set of failure types and ψ be the failure assignment function with $\psi(\sigma_{uo}) = \psi(\sigma_i) = \emptyset, i = 1, 2, 3, \psi(\sigma_{f1}) = F_1, \psi(\sigma_{f2}) = F_2$. From the first step in Algorithm 1,

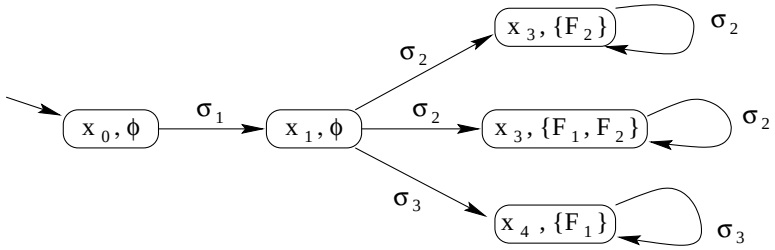

Figure 3.2: Diagram of G_o

we can derive G_o from G, which is shown in Figure 3.2. The strict composition of G_o with itself, $G_d = G_o||G_o$, is derived from the second step in Algorithm 1, which is shown in Figure 3.3. In Figure 3.3, there is a self loop at the state $((x_3, \{F_2\}), (x_3, \{F_1, F_2\}))$. So from

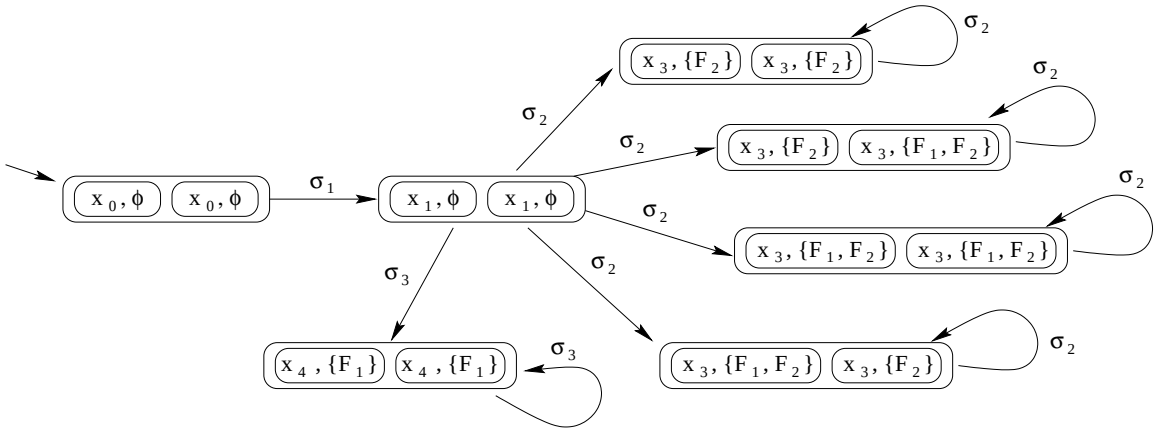

Figure 3.3: Diagram of G_d

the last step in Algorithm 1 we know the system G is not diagnosable.

Now suppose we need not distinguish the failure type F_1 from the type F_2. Then by letting $F_2 = F_1$ in Figure 3.3 and deleting some redundant states, we can obtain the corresponding G_d for the modified system. The resulting G_d is omitted here. In the modified G_d, there does not exist any cycle as stated in step 3 of Algorithm 1. So we know the modified system is diagnosable.

3.4 Conclusion

In this chapter, an algorithm is provided for testing the diagnosability of discrete event systems. Compared to the existing testing method in [98], our algorithm does not require

the construction of a diagnoser for the system. The complexity of our algorithm is of 4th order in the number of states of the system and linear in the number of failure types of the system, whereas the complexity of the testing method in [98] is exponential in the number of states of the system and doubly exponential in the number of failure types of the system.

Chapter 4

Modeling Discrete Event Systems with Faults using a Rules Based Modeling Formalism

4.1 Introduction

Failure refers to a deviation from a specified behavior of the system (or a component of the system) for a bounded or unbounded period of time. A stuck-close valve, decrease in the efficiency of a heat exchanger, abnormal bias in the output of a sensor, and leakage in pipelines are examples of failures. A major factor that motivates research on failure diagnosis arises from the fact that failures are inevitable in the existing industrial environment. Given the complex interactions between components, sub-systems, and processes, a system failure is considered to be a normal occurrence [85], or an inherent characteristic of systems. In large and complex systems, failure diagnosis is a critical task. This problem has received a great deal of attention in the literature of various domains including that of discrete event systems (DESs) [88] in [14, 39, 64, 98, 93]. A state based DES approach to failure diagnosis was proposed in [67], and further treated in [5]. Sampath *et al.* have proposed a formal language framework for studying the diagnosability properties of untimed discrete event systems [98, 97, 95].

In this chapter, an application of the rules based modeling formalism [12] to modeling DESs with faults is presented. We adopt an *input/output* view of the system, wherein the system receives inputs and generates outputs. The input signals of the system constitute

the independent variables, and output signals the dependent variables which are a function of the independent variables and other dependent variables. All signals in the system are assumed to be binary valued. Input signals change their values depending only on their own present values; whereas output signals change their values based on their own values and the values of other signals.

Faults in the system occur because of the malfunction of actuator or sensor signals in the system, or because of the malfunctions in the system itself. Examples of the latter kind of fault are those equipment failures which occur spontaneously, such as a tank beginning to leak, the power supply of a PLC failing, and so on. These can occur without regard to the value of any other signal in the system. Hence equipment failures are a type of input signals. On the other hand, faults can occur in actuators which are the inputs applied to the system; and they can occur in sensors which record the observable part of the system outputs. Such faults usually depend on the state of other signals in the system, and hence are a type of output signals. As an example consider the operation of actuators in a hydraulic system: When an actuator is being turned on, it can get stuck in the off position not permitting the flow of the fluid through it. This kind of fault is termed as a *stuck closed* fault. There is a corresponding *stuck open* fault when the actuator, once open, cannot be mechanically shut off. A similar type of fault can occur in the sensors: When a sensor is actuated it can get stuck in this position, causing a *stuck up* position fault. A corresponding *stuck down* fault can occur when the sensor, once it reaches the unactuated position, cannot change its value. It can be seen that the occurrence of a stuck-signal fault depends on the present value of the corresponding signal.

In order to obtain models of the system with faults, we apply the framework of [12], which relies on establishing boolean enabling or guard conditions for each event of the DES. An event is a transition of an input or an output signal from one binary value to another. We start by establishing the *initial conditions* of the system signals. Next, for each of the input and output events of the system, including the fault events, we obtain *event occurrence rules*. For output events these are boolean constraints over values of all the signals of the system, while for input events these are constraints only on the input signals themselves. Weights are associated with signals which indicate the degree to which they influence the occurrence of output events. In order to model failure prone systems, faults are incorporated into the event occurrence rules [32, 2], and from a modeling standpoint they are treated just the same as any other signal in the system. A binary value fault signals is introduced to

model presence or absence of a certain fault. Addition of new fault signals requires new rules for the added fault events, and modification of rules of existing non-faulty events, by appropriately weakening their guard conditions.

The representation of a system with faults, in the rules based modeling formalism, is polynomial in the size of signals and faults. The compactness of this model, together with its intuitive nature, makes it user-friendly, less error-prone, more flexible, easily scalable, and provides canonicity of representation for models of systems with faults.

The rest of the chapter is organized as follows: In Section 2, the preliminaries related to modeling DESs are discussed. This is followed by an example drawn from process control in Section 3. In Section 4, the types of fault which can occur in a system along with their representation in the modeling formalism is discussed. This is illustrated through the earlier process control example in Section 5. Section 6 provides conclusions and directions for future research.

4.2 Notation and Preliminaries

The possible sequencing of input and output events of a DES can be represented by a set of interacting automata. An overview of the automata based model of DES follows. Let Σ denote the finite set of events. A concatenation of events forms a *trace*. A *language* is a collection of traces. Let Σ^* be the set of all finite traces of events of Σ including the zero length trace ϵ. A language is thus a subset of Σ^*.

A discrete event system is represented by a finite collection of extended automata (i.e., automata with enabling guard conditions on transitions) G_i indexed by i. An automaton transitions from one state to another in response to the execution of an event provided a certain guard condition is satisfied. Formally an extended automaton is a 5-tuple: $G_i = (X_i, \Sigma_i, E_i, x_i^0, X_i^m)$; where X_i is the finite set of states, Σ_i is the finite set of events, E_i is the finite set of state transitions, $x_i^0 \in X_i$ is the initial state, and $X_i^m \in X_i$ is the set of final states. Each transition $e \in E_i$ is a quadruple of the form, $e := (x^e, \sigma^e, \mathcal{P}^e(\Pi_i X_i), y^e)$, where $x^e \in X_i$ is the state where the transition is executed, $\sigma^e \in \Sigma_i$ is the event label of the state transition, $y^e \in X_i$ is the state resulting from the execution of the transition, $\mathcal{P}^e(\Pi_i X_i)$ is the guard condition–a predicate over the states of the interacting automata–which must be satisfied for the transition to occur. A transition is enabled at a state when the associated guard condition evaluates to `true`.

In order to obtain the overall model of the system, the synchronous composition of automata, presented in [36], is extended to that for extended automata. Without loss of generality, we define the synchronous composition of two extended automata, $\{G_i := (X_i, \Sigma_i, E_i, x_i^0, X_i^m)\}_{i=1,2}$. The synchronous composition of G_1 and G_2, denoted $G_1 \| G_2$, is the automaton (X, Σ, E, x^0, x^m), where $X := X_1 \times X_2$, $\Sigma := \Sigma_1 \cup \Sigma_2$, $x^0 := (x_1^0, x_2^0)$, and the set of transitions $E = E_\alpha \cup E_\beta \cup E_\gamma$, where:

$$
\begin{aligned}
E_\alpha &:= \{((x_1^e, x_2^e), \sigma^e, \mathcal{P}^e, (y_1^e, y_2^e)) \mid \\
&\quad \exists (x_1^e, \sigma^e, \mathcal{P}_1^e, y_1^e) \in E_1, (x_2^e, \sigma^e, \mathcal{P}_2^e, y_2^e) \in E_2 \text{ s.t. } \mathcal{P}_1^e \wedge \mathcal{P}_2^e = \mathcal{P}^e\} \\
E_\beta &:= \{((x_1^e, x_2^e), \sigma^e, \mathcal{P}^e, (y_1^e, x_2^e)) \mid \\
&\quad \exists (x_1^e, \sigma^e, \mathcal{P}^e, y_1^e) \in E_1 \text{ s.t. } \sigma^e \in \Sigma_1 - \Sigma_2\} \\
E_\gamma &:= \{((x_1^e, x_2^e), \sigma^e, \mathcal{P}^e, (x_1^e, y_2^e)) \mid \\
&\quad \exists (x_2^e, \sigma^e, \mathcal{P}^e, y_2^e) \in E_2 \text{ s.t. } \sigma^e \in \Sigma_2 - \Sigma_1\}.
\end{aligned}
$$

E_α is the set of transitions which occur synchronously with the participation of both G_1 and G_2, whereas E_β and E_γ, respectively, are the set of transitions that occur asynchronously with the participation of G_1 and G_2 only.

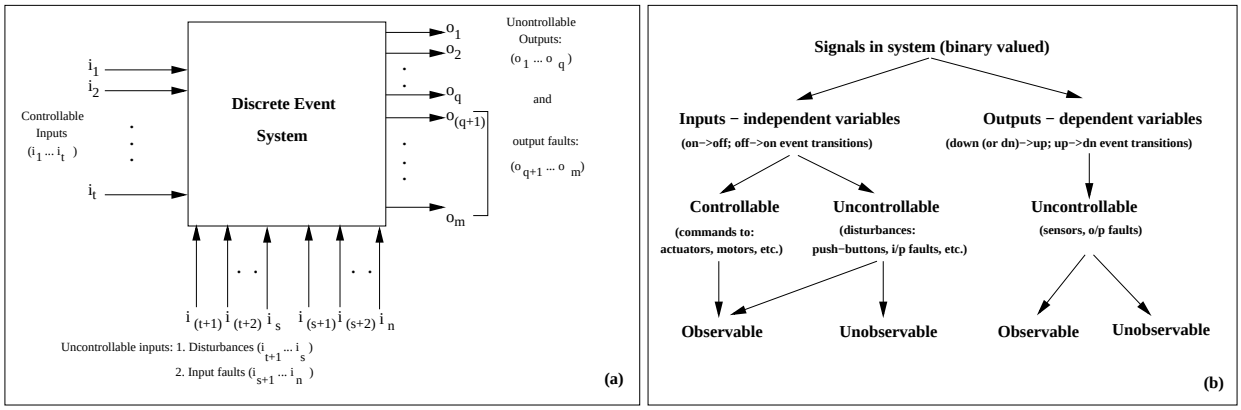

Figure 4.1: Input-Output view of a discrete event system

In the rules based modeling formalism of [12] the signals present in the system are partitioned into two sets: input signals and output signals. A block diagram of a discrete event system is shown in Figure 4.1(a), which has n input and m output signals. Input signals constitute the *independent variables* of the system. Their transitions occur depending solely on their own present value, i.e. their occurrence is not influenced by any other signal in

the system. The input signals are further divided into controllable and uncontrollable parts depending on whether an external agent can disable them or not. Controllable input events are the command signals sent to actuators, valves, motors of the system, in response to which the system evolves. Uncontrollable input events are those input signals which can neither be disabled nor enabled by any controller devised for the system. These can be operator push buttons, switches, as well as those faults occurring in the system which are not determined by the value of any other signal in the system. Output signals constitute the *dependent variables* of the system. Their transitions occur depending on their own values and values of other signals in the system. Output events are generated by sensors, as well as by those faults occurring in the system which are dependent on the values of other signals of the system.

There are t controllable inputs, $s-t$ uncontrollable disturbance inputs, and $n-s$ uncontrollable fault inputs. Output signals include q non-faulty output signals, and $m-q$ fault signals. All of the m output signals are considered to be uncontrollable. Only those output signals which have corresponding sensors connected for monitoring their event transitions can be observed. The rest are termed to be unobservable. All controllable events in the system can be monitored, and are termed as observable events as well. In Figure 4.1(b), all events in the system are further classified according to the properties of these events.

The rules based modeling formalism introduced in [12] is reviewed next. We model systems involving non-discrete variables for which a discrete event system abstraction is being sought, as is often the case in process control systems. Such systems possess signals that take values in a continuum such as flow rates, temperatures. However, only their discrete values are of interest for the purposes of modeling and analysis.

The rules based modeling formalism applies to systems for which the system inputs and outputs are binary valued, and all the system states are determined by the current values of the input and output signals of the system. A model in this formalism consists of:

1. *Initial Conditions:* The system starts out with certain initial values of all its signals which is captured by the initial conditions in our modeling formalism corresponding to the initial values of inputs and outputs of the system. This initial state is commonly the state when parts in system are least and all the actuators are turned off.

2. *Event occurrence rules:* For the input events, it suffices to know just the present value of the input signal to determine the next possible input event. This is because input signals alternate between their off and on values regardless of the values of any other

signals in the system. The occurrence of output events, however, is initiated by prior occurrence of other events. Also, in most physical systems, there is a relation between the way sensors are arranged physically and the order in which their sensed values change. Such signal dependencies are captured through event occurrence rules. There is one such rule per event. The consequent of each rule is an event, whereas the antecedent is a boolean formula over the signals present in the system.

For the p'th input signal i_p, its event occurrence rule takes the following form:

$$Rule_p^{on} : \overline{i_p} \Rightarrow i_p on;$$
$$Rule_p^{off} : i_p \Rightarrow i_p off;$$

These rules are termed as *default* since they capture the default constraint that signals alternate between their *on* and *off* values. For a system with n input signals, there will be $2n$ such default rules corresponding to each of the $2n$ input events in the system.

The rules for output events take on the following form for a system with m output signals, and n input ones:

$$\begin{aligned}
Rule_1^{up} &: f_1^{up}(i_1^{w_1}, ..., i_n^{w_n}, o_1, ..., o_j, o_{j+1}^{w_{j+1}}, ..., o_m^{w_m}) \Rightarrow o_1 up, \\
Rule_1^{dn} &: f_1^{dn}(i_1^{w_1}, ..., i_n^{w_n}, o_1, ..., o_j, o_{j+1}^{w_{j+1}}, ..., o_m^{w_m}) \Rightarrow o_1 dn, \\
&\vdots \\
Rule_m^{up} &: f_m^{up}(i_1^{w_1}, ..., i_n^{w_n}, o_1, ..., o_j, o_{j+1}^{w_{j+1}}, ..., o_m^{w_m}) \Rightarrow o_m up, \\
Rule_m^{dn} &: f_m^{dn}(i_1^{w_1}, ..., i_n^{w_n}, o_1, ..., o_j, o_{j+1}^{w_{j+1}}, ..., o_m^{w_m}) \Rightarrow o_m dn;
\end{aligned}$$

where $f_1^{up/dn}$, ..., $f_m^{up/dn}$ are the boolean formulae, consisting of those combinations of the input signals i_1, ..., i_n, and output signals o_1,..., o_m, which when `true` can result in the enablement of an output event. Note that each input signal and certain output signals $(o_{j+1}, .., o_m)$ are superscripted with the weight with which they influence the associated output event. Some other output signals $o_1, .., o_j$ are not weighted, or implicitly, their weights are simply the default value 1.

The antecedent of each rule is written in the disjunctive form, where each disjunct itself is the conjunct of three terms. The three terms within each disjunct are of the form:

(a) $f_l^e(i_1^{w_1}, ..., i_n^{w_n}, o_{j+1}^{w_{j+1}}, ..., o_m^{w_m})$:= a weighted boolean formula over signals representing an enabling condition for the consequent output event.

(b) $f_l^d(i_1^{w_1}, ..., i_n^{w_n}, o_{j+1}^{w_{j+1}}, ..., o_m^{w_m})$:= a weighted boolean formula over signals representing a disabling condition for the consequent output event.

(c) $f_l^s(i_1, ..., i_n, o_1, ..., o_m)$:= a boolean formula over unweighted input and output signals representing an enabling condition for the consequent output event.

For $Rule_k^{up/dn}$, the antecedent of the rule then takes the following form:

$$\bigvee_{l=1}^{N_k^{up/dn}} \{[\underbrace{w(f_l^e(i_1^{w_1}, ..., i_n^{w_n}, o_{j+1}^{w_{j+1}}, ..., o_m^{w_m}))}_{\text{enabling signals' weight}} > \underbrace{w(f_l^d(i_1^{w_1}, ..., i_n^{w_n}, o_{j+1}^{w_{j+1}}, ..., o_m^{w_m}))}_{\text{disabling signals' weight}}]$$
$$\wedge [\quad \underbrace{f_l^o(i_1, ..., i_n, o_1, ..., o_m)}_{\text{condition on unweighted signals}} \quad]\}$$

For a system with m output signals, there are $2m$ event occurrence rules corresponding to each of the $2m$ output events in the system.

In order to evaluate the weight of a weighted boolean formula, either a *minimum* or a *summation* operation is used depending on whether the combination of the input signals is an AND or an OR. When the composition is based upon minimum and summation operations, the weight of a weighted boolean formula is defined inductively as follows:

$$w(i_1^{w_1} \wedge i_2^{w_2}) = \min(w(i_1), w(i_2)) = \min(w_1, w_2); \quad w(i_1^{w_1} \vee i_2^{w_2}) = w(i_1) + w(i_2) = w_1 + w_2.$$

Thus for example,

$$w((i_1^{w_1} \wedge i_2^{w_2}) \vee i_3^{w_3}) = w(i_1^{w_1} \wedge i_2^{w_2}) + w(i_3) = min(w(i_1), w(i_2)) + w(i_3).$$

Note that the choice of minimum and summation based combination of weights is application dependent, and its semantics may be changed from application to application without changing the syntax of the modeling formalism.

Once the rules for all the output and input events of the system have been obtained, the algorithm of [12] may be used for translating the rules based model into an equivalent automata model of the system. For this, the system is first represented as a composition of a set of interacting 2-state extended automata, one automata for each of the binary valued signals of the system. The enabling conditions present in the event occurrence rules appear as guards for the event transitions in the 2-state extended automata models. A composition of the interacting extended automata yields the desired automaton model, in which the transitions with guard conditions as `false` are simply deleted.

4.3 Motivating Example: A Tank System

Consider a tank filling system whose schematic is shown in Figure 5.1. It has one filling

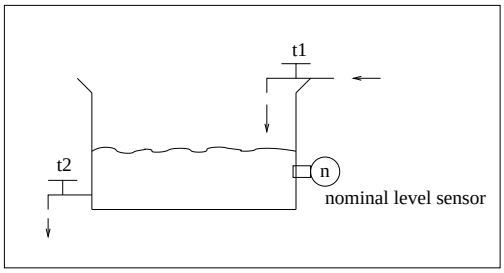

Figure 4.2: Tank system schematic

tap *t1*, one draining tap *t2*, and a nominal level sensor *n*. The signals in the system are *t1, t2, n*, and the events that can occur in this system are *t1on, t1off, t2on, t2off, nup, ndn*. Assume that the filling rate of tap *t1* has the flow value of +10, while that of the draining is +1.

The model in the rules based formalism, consisting of initial conditions and event occurrence rules, is given in Figure 5.2. Note that $Rule_1^{up}$ simply states that for the event *nup* to occur, the system should be in a state where the filling rate exceeds the draining rate, and the level sensor is down. $Rule_1^{dn}$ is similar, and other rules are default ones.

1. Initial conditions: $t1 = t2 = [\texttt{off}]$; $n = [\texttt{down}]$.

2. Event occurrence rules: Since there is 1 sensor signal, n, it has 2 sensor events associated with it, i.e. nup/ndn, and so there are 2 rules for these 2 events. In addition there are 4 default ones for the 2 input signals $t1, t2$.

$$Rule_1^{up} : ([t1^{+10}] > [t2^{+1}]) \wedge [\overline{n}] \Rightarrow nup;$$
$$Rule_1^{dn} : ([t2^{+1}] > [t1^{+10}]) \wedge [n] \Rightarrow hup;$$
$$Rule_2^{on} : \overline{t1} \Rightarrow t1on; \qquad Rule_2^{off} : t1 \Rightarrow t1off;$$
$$Rule_3^{on} : \overline{t2} \Rightarrow t2on; \qquad Rule_3^{off} : t2 \Rightarrow t2off.$$

Figure 4.3: Rules based model of the tank system without faults

An equivalent automata model, obtained using the algorithm presented in [12], is shown in Figure 4.4. The automaton shown in Figure 4.4(d) is obtained by taking a synchronous

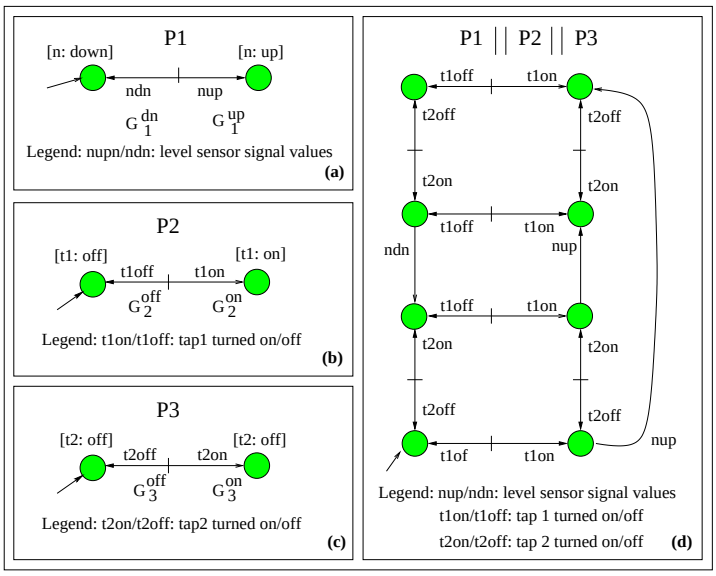

Figure 4.4: Automaton model of tank system without faults

composition of the extended 2-state automata models of the taps, *t1* and *t2*, (Figure 4.4(a) and (b) respectively), and that of the water level sensor, n, (Figure 4.4(c)). Note that the antecedent of the rules appear as guards in the 2-state extended automata models. For example, G_1^{up} is the weighted boolean formula $([t1^{+10}] > [t2^{+1}]) \wedge [\overline{n}]$, which is the antecedent of $Rule_1^{up}$ of Figure 5.2. Transitions of the composed automata, in which the guards evaluate to `false`, have been omitted.

4.4 Modeling failures in the rules based formalism

In order to obtain event occurrence rules for a system with faults, we examine the possible kinds of faults, and also the manner in which they are represented in the rules based modeling formalism.

4.4.1 Signal and system faults

The kinds of faults which can occur in the type of discrete event systems we consider include, the stuck-signal faults, and the system or equipment faults.

1. **Stuck-signal faults:** These faults occur when any of the actuators or sensors in the system, owing to mechanical, electrical, or electromagnetic interference problems, gets stuck in a particular position with its logic status becoming either `true` or `false` permanently, until a recovery occurs through a repair or a replacement. When an *actuator* gets stuck in the *on/off* position such fault signals are denoted by *so*(stuck open)/*sc*(stuck closed) respectively; where the associated fault events are denoted by *soF/scF* and the recovery events by *soR/scR*, respectively. In the tank system the filling tap *t1* may be prone to an stuck open fault signal denoted by *t1so*, with the fault event denoted as *t1soF* and the recovery event denoted as *t1soR*. When the fault event *t1soF* occurs, filling will continue to occur even after the command to switch off the tap has been given, unless the fault recovery event occurs.

   When a *sensor* gets stuck in the *up/dn(down)* position such fault signals are denoted by *sup*(stuck up)/*sdn*(stuck dn) respectively; whereas the associated fault events are denoted by *supF/sdnF*, and the recovery events by *supR/sdnR*, respectively. The tank system in Figure 5.1 has a level sensor n which may be prone to a stuck up fault signal, denoted by *nsup*. This signal has two events: the fault event *nsupF*, and the corresponding recovery event *nsupR*.

   It should be noted that stuck-signal faults are a type of output signals since they are dependent on values of signals prone to stuck-signal faults. Referring to Figure 4.1(a), it can be seen that of the m output signals, $m - q$ are fault signals.

2. **System/equipment faults:** Apart from faults of the signals there are faults of systems and its components. Certain fault signals such as equipment failures, power disruptions, system software crashes, etc., affect the entire system. These can occur spontaneously in the system depending only on their own values, not those of any other signal in the system. They are thus independent variables and form part of the inputs to the system. In the tank system of Figure 5.1, a *leakage* fault signal, which causes the fluid levels to drop in the tank, is an example of a system/equipment fault. The events of the leakage fault are *leakageF* and *leakageR*, denoting leakage fault and recovery from leakage events.

   It should be noted that system faults are a type of input signals since they are independent of values of other signals. Referring to Figure 4.1(a), it can be seen that of the n input signals, $n - s$ are fault signals.

4.4.2 Permanent and intermittent faults

Another categorization of faults arises from the manner in which faults are reset after they occur.

1. **Permanent faults:** If the recovery event occurs only due to a repair/replacement of the fault, then the fault is regarded as a *permanent fault*.

2. **Intermittent faults:** If the recovery event can occur either spontaneously or through repair/replacement, then the fault is regarded as an *intermittent fault*. Example is a loose wire that makes and breaks contact spontaneously.

It is important to distinguish between these two types of faults, since the intermittent fault spontaneous recovery events, which tend to be uncontrollable and unobservable, may the system to oscillate between non-faulty and fault states. Permanent faults, on the other hand, are associated with recovery events (repair/replacement) which are controllable and observable, and the system cannot spontaneously move from a fault state to a non-fault one.

4.4.3 Rules for fault events

1. **Stuck-signal faults:** Stuck-signal faults occur in both actuators and sensors when the value of the signal gets stuck at a certain logic level. The rules for both these kinds of signals are discussed next:

   - *Stuck actuator signal faults:* If the r'th actuator, A_r, is prone to both the *stuck open* fault which occurs only after the actuator is already in the on condition, and the *stuck closed* fault which occurs only when the actuator is already in the off condition, then the rule for the occurrence of these stuck actuator signal faults is written as:

   $$Rule^{on}_{A_rso} : A_r \wedge \overline{A_rso} \Rightarrow A_rsoIF; \qquad Rule^{off}_{A_rso} : A_rsoIF \Rightarrow A_rsoR;$$
   $$Rule^{on}_{A_rsc} : \overline{A_r} \wedge \overline{A_rsc} \Rightarrow A_rscIF; \qquad Rule^{off}_{A_rsc} : A_rscIF \Rightarrow A_rscR.$$

   Here recovery from the fault is considered to spontaneously occur without regards to the value of any other signals in the system. In case the recovery is initiated by other signals in the system they will appear in the antecedent of the rule.

   As an example consider the tank system of Figure 5.1 with a faulty inlet tap *t1*, prone to a stuck open fault. Assuming that no recovery is possible from the stuck

open fault event, the rules for the fault events *t1soF* and *t1soR* are:

$$Rule^{on}_{t1so} : t1 \wedge \overline{t1so} \Rightarrow t1soF; \qquad Rule^{off}_{t1so} : \texttt{false} \Rightarrow t1soR.$$

- *Stuck sensor signal faults:* If the r'th sensor in a system, S_r, is prone to both the *stuck up* fault which occurs only after it has been actuated, and the *stuck down* fault which occurs only when the sensor is not actuated, then the rule for the occurrence of these stuck sensor signal faults is written as:

$$Rule^{up}_{S_rsup} : S_r \wedge \overline{S_rsup} \Rightarrow S_rsupF; \qquad Rule^{dn}_{S_rsup} : S_rsup \Rightarrow S_rsupR;$$
$$Rule^{up}_{S_rsdn} : \overline{S_r} \wedge \overline{S_rsdn} \Rightarrow S_rsdnF; \qquad Rule^{dn}_{S_rsdn} : S_rsdn \Rightarrow S_rsdnR.$$

Stuck-signal failures of signals may have a non-unity weight assigned to them.

2. **System or equipment faults:** The rules for the t'th equipment fault, E_t, is expressed in the rules based modeling formalism as:

$$Rule^{on}_{E_t} : \overline{E_t} \Rightarrow E_tF; \qquad Rule^{off}_{E_t} : E_t \Rightarrow E_tR.$$

Here the recovery from the fault is considered to be occur independently of the values of any other signals in the system. In case the recovery is initiated by other signals in the system, they will appear in the antecedent of the rule.

In the process control system of Figure 5.1, a *leakage* fault, which causes the fluid level to drop, is an example of a system fault. Now in addition to the existing rules shown in Figure 4.7, an additional pair of rules $Rule^{on/off}_6$, is defined to account for this newly added fault signal:

$$Rule^{on}_6 : \overline{leakage} \Rightarrow leakageF; \qquad Rule^{off}_6 : leakage \Rightarrow leakageR.$$

3. **Intermittent faults:** If the r'th signal/equipment, A_r, is prone to an intermittent fault, recovery from such a fault can occur either spontaneouly (A_rIR) or by repair/replacement of the faulty device (A_rR).

$$Rule^{on}_{A_rso} : A_r \wedge \overline{A_rso} \Rightarrow A_rsoIF;$$
$$Rule^{off}_{A_rso} : (A_rso) \wedge (A_rR \vee A_rIR) \Rightarrow A_rsoR;$$
$$Rule^{on}_{A_rsc} : \overline{A_r} \wedge \overline{A_rsc} \Rightarrow A_rscIF;$$
$$Rule^{off}_{A_rsc} : (A_rsc) \wedge (A_rR \vee A_rIR) \Rightarrow A_rscR.$$

As an example consider the tank system of Figure 5.1 with a faulty inlet tap *t1*, prone to a stuck open fault. The rules for the fault events *t1soF* and *t1soR* are:

$$Rule^{on}_{t1so} : t1 \wedge \overline{t1so} \Rightarrow t1soIF; \qquad Rule^{off}_{t1so} : (t_1so) \wedge (t_1R \vee t_1IR) \Rightarrow t1soR.$$

4.4.4 Extension of non-fault event rules to include fault conditions

In the presence of faults the guard conditions of the non-fault event rules are weakened by introducing additional disjunctive conditions under which the non-fault event can also occur. The antecedent of each rule now contains the disjunct of a group of terms which represent how the consequent event can occur under both non-faulty and faulty conditions.

Example of extension of rules under stuck-signal faults: Consider the tank system of Figure 5.1 with a faulty inlet tap *t1* having a filling rate of +10, prone to a stuck open fault, whose occurrence does not alter the non-faulty filling rate through the tap. The rule for the output level sensor event *nup* in the presence of the stuck open fault *t1so* is given in Figure 4.7 as:

$$Rule_1^{up} : \underbrace{([t1so^{+10} > t2^{+1}] \wedge [\overline{n}])}_{\text{fault conditions}} \bigvee \underbrace{([t1^{+10} > t2^{+1}] \wedge [\overline{n} \wedge \overline{t1so}])}_{\text{non-fault conditions}} \Rightarrow nup.$$

The antecedent of this rule is now a disjunct of two terms, the first one corresponds to the level sensor going up under the *t1so* fault; whereas the second one corresponds to the level sensor going up under the non-faulty condition, and is essentially the same as the corresponding antecedent for the system model without faults.

Note that if the actuator *t1* in the tank system of Figure 5.1 is subjected to a stuck open fault, the rate of filling may change from the normal value to some other value under faulty conditions. So, for example, if the filling rate changes from +10 to +7 in the presence of the *t1so* fault, then the output event occurrence rule becomes:

$$Rule_1^{up} : \underbrace{([t1so^{+7} > t2^{+1}] \wedge [\overline{n}])}_{\text{fault conditions}} \bigvee \underbrace{([t1^{+10} > t2^{+1}] \wedge [\overline{n} \wedge \overline{t1so}])}_{\text{non-fault conditions}} \Rightarrow nup.$$

Example of extension of rules under system/equipment faults: Owing to the modeling of the *leakage* fault signal, the rule for the filling event (*nup*) and for the draining event (*ndn*) are altered as well. For example, the new extended rule $Rule_1^{up}$, having a *t1soF* with a filling rate of +10, and a draining tap *t2* with a draining rate of +1, and the tank susceptible to a leakage fault having a draining rate of +3, is given by:

$$Rule_1^{up} : \underbrace{([t1so^{+10} > (t2^{+1} \vee leakage^{+3})] \wedge [\overline{n}])}_{\text{fault conditions}} \bigvee \underbrace{(([t1^{+10} > [t2^{+1} \vee leakage^{+3}])] \wedge [\overline{n} \wedge \overline{t1so}])}_{\text{non-fault conditions}}$$
$$\Rightarrow nup.$$

4.4.5 Fault signal automata models

In order to account for the faults in a system, additional 2-state extended automata

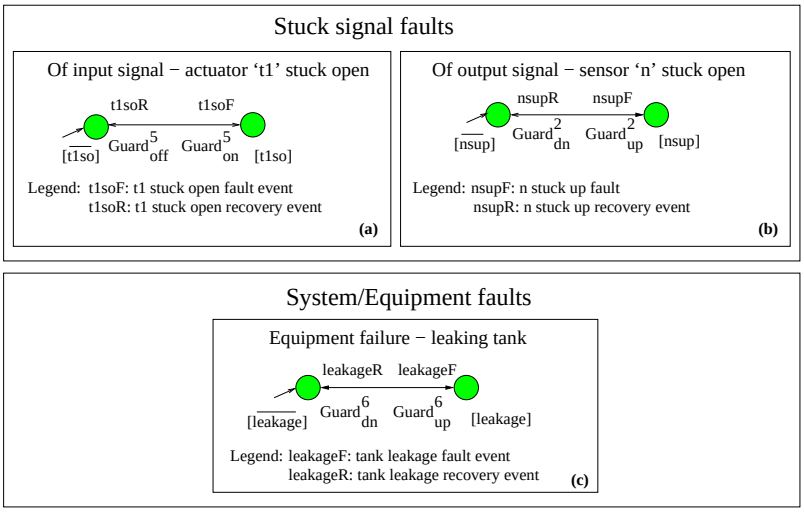

Figure 4.5: Automata models for signal and system faults

models are drawn with their states representing the faulty and non-faulty conditions. These extended automata are composed with the rest of the extended automata to obtain a single automaton model of a DES that is subjected to failures. The two fault scenarios considered in the rules based modeling formalism are stuck-signal value faults and the system/equipment faults.

For the tank system of Figure 5.1, the corresponding automata models for the example faults which can occur in the system are shown in Figure 4.5. The 2-state extended automata, which model the *stuck open* fault which may occur in the actuator *t1*, and the *stuck close* fault which may occur in the level sensor *n*, are shown in Figure 4.5(a), (b) respectively, along with their corresponding recovery events. In Figure 4.5(c), a 2-state automata model is given for a type of system fault, the *leakage* fault, having as its states no-leakage ($\overline{[leakage]}$) and leakage ($[leakage]$), with transitions between these states on events leakage-fault ($leakageF$) and leakage-recovery ($leakageR$), respectively. The antecedent of the event occurrence rules associated with the events appear as guard conditions in the 2-state extended automata model. Refer to Figure 4.7 for the guards for the fault events $t1soF, t1soR, nsupF$, and $nsupR$, and to Section 4.4 above for the guards for the fault events $leakageF$, and $leakageR$. Only when the guard condition evaluates to `true`, is the corresponding event transition in the automaton permitted to occur.

4.5 Application: modeling tank systems with faults

In order to illustrate the modeling of systems with faults in the rules based formalism, we introduce certain fault conditions in the tank system of Figure 5.1, and obtain an automaton and an rules based model for it. Assume that the filling actuator *t1* can get stuck in the open position, represented by an output fault signal, *t1so*, and the level sensor can get stuck in the `true` state, represented by an output fault signal, *nsup*. The complete set of events that can occur in this system are *t1on, t1off, t2on, t2off, nup, ndn, t1soF, t1soR, nsupF, nsupR*. Here *t1soR* is the *t1* stuck open recovery event and *nsupR* is the *n* stuck up recovery event. Assume that the filling rate of tap *t1* has the flow value of +10, even in the presence of the *t1so* fault, while the draining rate of the tap *t2* is +1.

We present an automaton model of the system and next present the rules based model which contains as much information as the more complex automaton model, but is compact and simpler to obtain and debug. The automaton model of the system is shown in Figure 4.6,

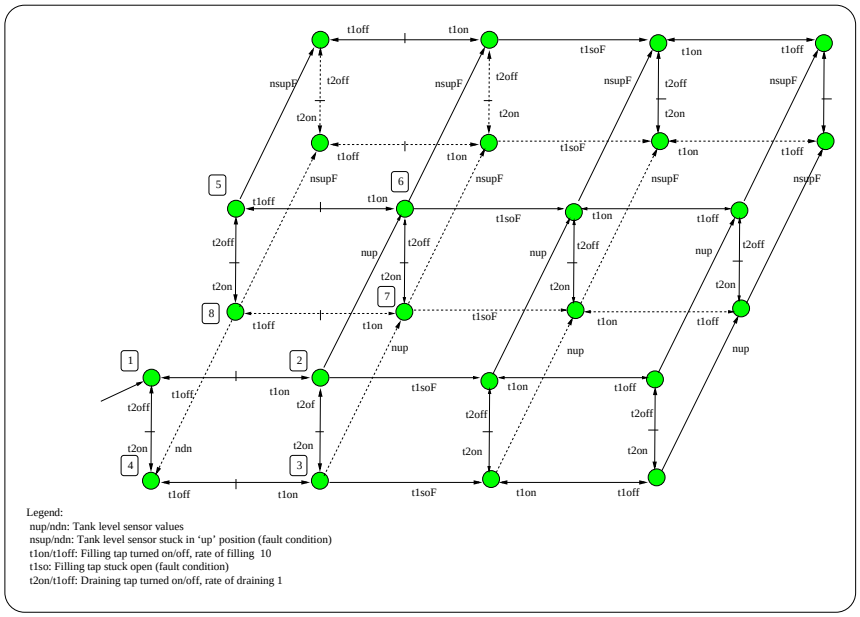

Figure 4.6: Automaton model of the tank system with faults

and has a total of 24 states, of which the states marked $1 \ldots 8$ represent the non-faulty states of the system. State 1 is the initial state in which the level sensor is low, and both the filling and draining taps are turned off. When the filling tap *t1* is turned on, *t1on*, then there is a transition to State 2, where the level of fluid in the tank rises, until either the *nup* event of

the sensor occurs taking the system to State 6, or the event *t1off* occurs causing the system to return to the initial state. Transitions to other states are drawn out in a similar fashion. In all the states where the tap *t1* is turned on, it can get stuck in this position, a *t1so* fault signal, and is indicated by a transition on the event *t1soF*. Once that event occurs commands of *t1on/t1off* have no effect on the system state, and the tap *t1* continues to permit fluid flow through it regardless of the control commands applied to it. A corresponding recovery event *t1soR* is defined which returns the system to the non-faulty state of the tap *t1*. Also, in all the states where the level sensor n is `true`, i.e, *nup*, the sensor can get stuck in this position, indicated by the occurrence of a fault signal *nsup* event *nsupF*, after which the sensor event *ndn* cannot occur even if the fluid level in the tank becomes low, until the recovery event *nsupR* occurs. We assume for this example that recovery events are infeasible (by disallowing any repair or replacement). So although recovery events $t1soR$ and $nsupR$ are defined, no transitions are feasible on such events in this example.

The model of the tank system with faults in the rules based modeling formalism is shown in Figure 4.7.

1. Initial conditions: $t1 = t2 = t1so = nsup = [\texttt{off}]; n = [\texttt{down}]$.

2. Event occurrence rules:

$$Rule_1^{up} : ([t1so^{+10} > t2^{+1}] \wedge [\overline{n}]) \bigvee ([t1^{+10} > t2^{+1}] \wedge [\overline{n} \wedge \overline{t1so}]) \Rightarrow nup$$
$$Rule_1^{dn} : ([t2^{+1} > t1so^{+10}] \wedge [n]) \vee ([t2^{+1} > t1^{+10}] \wedge [n \wedge \overline{t1so}]) \Rightarrow ndn;$$
$$Rule_2^{on} : \overline{t1} \Rightarrow t1on; \qquad Rule_2^{off} : t1 \Rightarrow t1off;$$
$$Rule_3^{on} : \overline{t2} \Rightarrow t2on; \qquad Rule_3^{off} : t2 \Rightarrow t2off;$$
$$Rule_4^{up} : n \wedge \overline{nsup} \Rightarrow nsupF; \qquad Rule_4^{dn} : \texttt{false} \Rightarrow nsupR;$$
$$Rule_5^{on} : t1 \wedge \overline{t1so} \Rightarrow t1soF; \qquad Rule_5^{off} : \texttt{false} \Rightarrow t1soR.$$

Figure 4.7: Rules based model of the tank system with faults

Note that the initial condition now includes conditions on both non-fault and fault signals. The rules are self-explanatory, and this rules based model contains the same amount of information as the more complex automaton model shown in Figure 4.6. As might be evident, attempting to obtain an automaton model for the tank system in the presence of actuator and sensor faults is not an easy task. In contrast, modeling the tank system prone

to failures by applying the procedure outlined in Section 4.4, yields a compact rules based model as is evidenced from Figure 4.7. If needed, this model can be automatically translated into an equivalent automaton model, resulting in the automaton of Figure 4.6.

4.6 Incorporating delay faults in the modeling formalism

In any real world system the timing of events occurring within it is of considerable significance, and in many cases if an event occurs either too soon or too late the system might not function properly. For example, in an assembly line if the interval between the arrival of parts at a buffer feeding a workstation is too short, then the buffer may get filled faster than the workstation can process parts, thereby forcing the operations of upstream machines to slow down or even stop completely. Also, if the inter-arrival times at the buffer are too large, then the workstation it feeds will get starved for parts, with machines further downstream possibly being affected as well. A similar scenario can occur in process control systems. For example, in the tank system shown in Figure 5.1, if the level of fluid in the tank is initially low, and the filling tap $t1$ is turned on, with or without the draining tap $t2$ being on as well, then in either case time bounds can be associated within which the level sensor signal should make a up-going transition. Hence, it is usually not sufficient to just specify that the logical properties of the system are not violated. We can also model the timing properties of signals which are of interest and report the occurrence of *delay faults* when the time-bounds associated with any event are violated.

4.6.1 Rules for timely occurrence/delay faults

The rules based modeling formalism can be extended to model real-time systems by including guard conditions that involve clock variables which monitor the time-bounds within which events should occur. For this, we may include certain "timed guards" along with the "untimed guards" in any rule.

Consider for example the "untimed rule" for an event σ with the untimed guard condition G_σ:

$$G_\sigma \Rightarrow \sigma.$$

Here G_σ is a predicate defined over the values of the signals of the system. In order to model

real-time behavior of the system, the untimed guard is augmented by certain timed guards defined over a certain set of clock variables C. Each clock in the set C is initialized to zero, and evolves at rate 1 as time elapses. These clocks keep track of the time since they were last reset due to the occurrence of a resetting event. The augmented rule specifies the condition under which the event σ should occur and is given by:

$$\exists i \in I : G^i_\sigma \wedge T^i_\sigma \Rightarrow \sigma, C^i_\sigma.$$

So, the event σ occurs when a guard $G^i_\sigma \wedge T^i_\sigma$ for some i in an index set I is satisfied. Here the untimed guard G_σ has been partitioned into I sub-guards $\{G^i_\sigma, i \in I\}$, i.e., $\bigvee_{i\in I} G^i_\sigma = G_\sigma$, and for each $i \in I$, T^i_σ is a timed guard defined over the set of clock variables. $C^i_\sigma \subseteq C$ is the set of clocks that are reset when the event σ occurs due to the satisfaction of the condition $G^i_\sigma \wedge T^i_\sigma$. If C^i_σ is not specified explicitly it implies that none of the clocks associated with the event σ are reset to their initial values.

Since the rule $\exists i \in I : G^i_\sigma \wedge T^i_\sigma \Rightarrow \sigma, C^i_\sigma$, specifies the condition for timely occurrence of the event σ, a delay fault is said to have occurred if σ occurs at an instance when the condition $\exists i \in I : G^i_\sigma \wedge T^i_\sigma$ is violated, or equivalently when the condition $\exists i \in I : G^i_\sigma \wedge \overline{T^i_\sigma}$ holds. Thus implicit in the rule of timely occurrence of the event σ, is another rule that captures a delay fault occurrence of σ, and is given by:

$$\exists i \in I : G^i_\sigma \wedge \overline{T^i_\sigma} \Rightarrow \sigma, C^i_\sigma.$$

The timed guard T^i_σ associated with the event σ is a predicate over the values of clocks, which is defined as follows. A timed guard φ is defined by the grammar

$$\varphi := x \leq c \mid c \leq x \mid x < c \mid c < x \mid \varphi_1 \wedge \varphi_2 \mid \neg\varphi,$$

where x is a clock in C and c is a rational constant. The set of all timed guards over the set of clocks C is denoted by $\Phi(C)$.

In the system, the input signals can be forced to occur at any time, meaning their timed guards are `true`. Also in the case of output signals, the stuck-signal and the system/equipment fault can occur at any time, and their timed guards are `true` as well. It is only the non-fault output signals such as the sensor signals that have non-trivial timeds guard associated with their events.

As an example consider the tank system of Figure 5.1 in which the combination of the inlet tap *t1* and the outlet tap *t2*, fill the tank within 4 time units. this can be captured in

a rule for nup as follows:

$$Rule_n^{up} : [(t1^{10} > t2^1) \wedge \overline{n}] \bigwedge [((t1 \wedge t2) \wedge [0 \leq c_{t1.t2} \leq 4]) \vee (\overline{t1 \wedge t2})] \Rightarrow nup.$$

Here $c_{t1.t2}$ is the clock that monitors the time elapsed since both $t1$ and $t2$ were switched on, and it is achieved by resetting $c_{t1.t2}$ in the rules for $t1on$ and $t2on$ as follows:

$$\begin{aligned} Rule_2^{on} &: \overline{t1} \Rightarrow t1on, \{c_{t1.t2}\}; \qquad & Rule_2^{off} &: t1 \Rightarrow t1off; \\ Rule_3^{on} &: \overline{t2} \Rightarrow t2on, \{c_{t1.t2}\}; \qquad & Rule_3^{off} &: t2 \Rightarrow t2off. \end{aligned}$$

We next model a more detailed example with delay faults occurring at multiple locations during a tank filling-draining process. Consider the tank system shown in Figure 5.1, with the nup event subject to a delay fault when both the filling and draining taps are turned on. For the purpose of illustrating the rules based framework under delay fault conditions only, we assume that no stuck-signal or equipment faults occur in the system.

In the tank system the filling rate of tap $t1$ is larger than the draining rate of tap $t2$. Hence, if initially the level of fluid in the tank is low, the level sensor nup event can occur either when only the filling tap $t1$ is turned on, or when both the filling and draining taps $t2$ are on. Assume that the time it takes to fill the tank when both the filling and draining taps are on is a maximum of 4 time-units, and 3.6 time-units when only the filling tap is on. On the other hand, it can be the case that the level sensor n is close to the switching level for nup at an instant when $t1$ is turned one, in which case the event nup occurs immediately. These scenarios set the bounds within with the event nup can occur. We associate a clock, $c_{t1.t2}$ for monitoring the time bound when both the taps are on, and $c_{t1.\overline{t2}}$ when tap $t1$ is on and tap $t2$ is off. The timing guard for $c_{t1.t2}$ is given by: $\{0 \leq c_{t1.t2} \leq 4\}$, and for $c_{t1.\overline{t2}}$ by $\{0 \leq c_{t1.\overline{t2}} \leq 3.6\}$. If the draining time is not of interest for delay fault monitoring, then no clock needs to be associated with the draining event. Alternately, a clock monitoring the ndn event can be associated with the time-bound $(0, +\infty)$.

The model of the tank system in the rules based modeling formalism is shown in Figure 4.8, where $Rule_1^{up}$ captures the timely occurrence of event nup.

Note that the clock $c_{t1.t2}$ is reset whenever there is a $t1on$ or $t2on$ event. The resets for $c_{t1.\overline{t2}}$ are obtained in a similar way, i.e., whenever there is a $t1on$ or a $t2off$ event.

1. Initial conditions: $t1 = t2 = t1so = nsup = [\texttt{off}]; n = [\texttt{down}]; c_{t1.t2} = 0.$

2. Event occurrence rules:

$$
\begin{aligned}
&Rule_1^{up} : [(t1^{+10} > t2^{+1}) \wedge \overline{n}] \ \bigwedge \\
&\qquad [(t2 \wedge (0 \leq c_{t1.t2} \leq 4)) \vee (\overline{t2} \wedge (0 \leq c_{t1.\overline{t2}} \leq 3.6))] \Rightarrow nup; \\
&Rule_1^{dn} : [t2^{+1} > t1^{+10}] \wedge [n] \Rightarrow ndn; \\
&Rule_2^{on} : \overline{t1} \Rightarrow t1on, \{c_{t1.t2}, c_{t1.\overline{t2}}\}; \\
&Rule_2^{off} : t1 \Rightarrow t1off; \\
&Rule_3^{on} : \overline{t2} \Rightarrow t2on, \{c_{t1.t2}\}; \\
&Rule_3^{off} : t2 \Rightarrow t2off, \{c_{t1.\overline{t2}}\}.
\end{aligned}
$$

Figure 4.8: Rules based model of the tank system with delay faults

4.6.2 Timed automaton model for timely occurrence/delay faults

The rules based model with timing guards can be represented using extended 2-state timed automata. An extended timed automaton is a 6-tuple:

$$G_i = (X_i, \Sigma_i, E_i, x_i^0, X_i^m, C),$$

where X_i is the finite set of states, Σ_i is the finite set of events, E_i is the finite set of state transitions, $x_i^0 \in X_i$ is the initial state, $X_i^m \subseteq X_i$ is the set of final states, C is a finite set of clocks. Each transition $e \in E_i$ is a 5-tuple of the form, $e := (x^e, \sigma^e, \mathcal{P}^e(\Pi_i X_i) \wedge \varphi^e, C^e, y^e)$, where $x^e \in X_i$ is the state where the transition is executed, $\sigma^e \in \Sigma_i$ is the event label of the state transition, $y^e \in X_i$ is the state resulting from the execution of the transition, $\mathcal{P}^e(\Pi_i X_i)$ is the untimed guard condition and $\varphi^e \in \Phi(C)$ is the timed guard condition—they must together be satisfied for the transition to occur, and $C^e \subseteq C$ is the set of clocks that get reset to zero when the transition occurs. A transition is enabled at a state when the associated guard condition evaluates to `true`.

By associating timing with the guards of an event representing the timely occurrence of the event, the delay fault information is included implicitly. Whenever the system is at a state from where an event is possible, by examining the timed portion of the guard and the occurrence time of event, the presence or absence of a delay fault can be identified.

The algorithm for automatically deriving an equivalent automaton model out of the

model in the proposed formalism is given next:

1. Obtain an untimed automaton, using the untimed portion of the rules as in [12].
2. For each transition on event σ, add the timing guard T_σ^i at the states where G_σ^i holds, and also incorporate the associated set of clocks C_σ^i that need to be reset.

For the tank example under consideration, the automata models of the level sensor n, containing the untimed guards are shown in Figure 4.9(a), and those for the events $t1on/off$, and $t2on/off$ are shown in Figure 4.9(b), (c). These extended automata are then composed

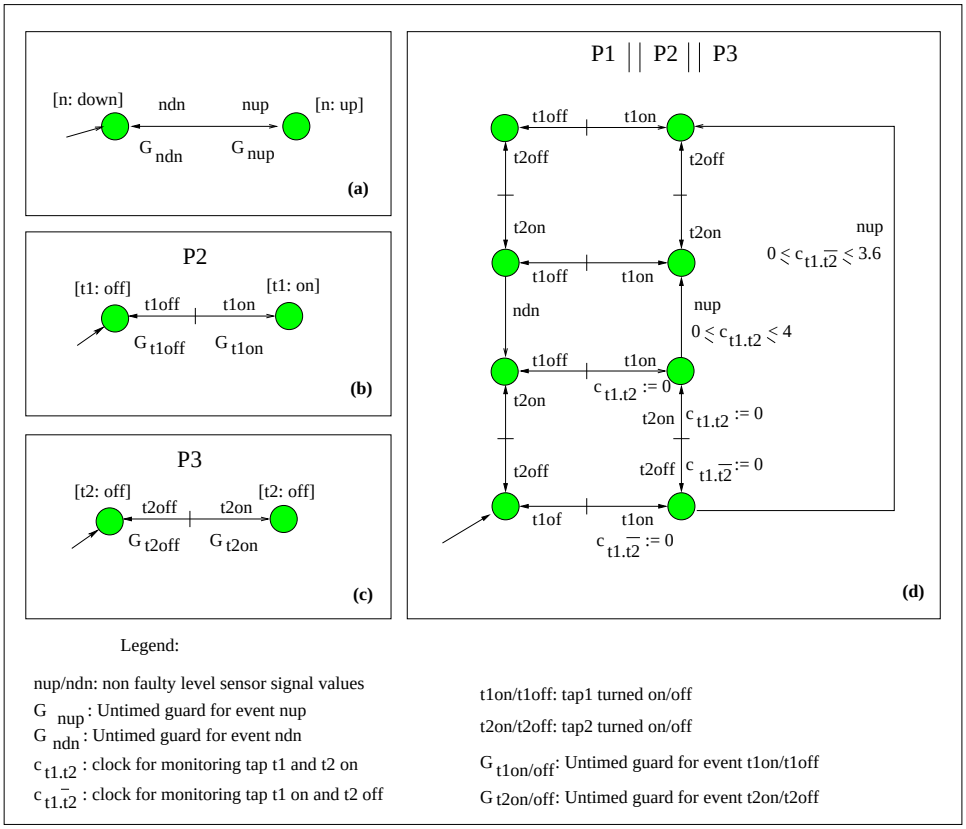

Figure 4.9: Timed automaton model for timely occurrence/delay faults

using synchronous composition, and the transitions whose untimed guards evaluate to `true` are retained in the final automaton. For the tank system this is shown in Figure 4.9(d). Now, the appropriate timed guards are added on appropriate transitions. The timed guard $(0 \leq c_{t1.t2} \leq 4)$ appears in Figure 4.9(d) on the *nup* event transition at the state where $t1$ and $t2$ are on; while the timed guard $(0 \leq c_{t1.\overline{t2}} \leq 3.6)$ on the *nup* event transition at

the state where $t1$ is on and $t2$ is off. The clocks $c_{t1.t2}$ and $c_{t1.\overline{t2}}$ are reset when events $t1on$ or $t2on$, and t1on or t2off occur respectively. This is the overall timed automaton of the system.

4.7 Conclusions

The rules based modeling formalism of [12] has been applied in order to obtain models of discrete event systems prone to failures. Stuck-signal faults, system/equipment faults and delay faults have been modeled in the rules based modeling formalism. This formalism presents a scalable as well as flexible alternative to the modeling discrete event systems prone to faults. Such models may be automatically converted to their equivalent automata models, for further analysis such as verification, diagnosis, and control. The technique has been demonstrated to work for a class of systems with discrete event system abstractions, comprising of boolean valued input/output signals. The formalism can be easily extended to include non-binary valued signals (more values means more states and transitions in the extended automata representation, and so more rules per signal). The rules based formalism being polynomial in the size of signals, provides a solution for the problem of state space explosion associated with automata models. The compact rules based model greatly aids rapid reconstruction and error-checking of the model, specially when elements are added, removed, or configured differently within the system.

Chapter 5

Diagnosis of Discrete Event Systems in Rules Based Model using First-order Linear Temporal Logic

5.1 Introduction

Detection and isolation of failures in large, complex systems is a crucial and challenging task. A failure is a deviation of a system from its normal or required behavior, such as occurrence of a failure event, or visiting a failed state, or more generally, violating a design specification. A stuck-close valve, decrease in the efficiency of a heat exchanger, abnormal bias in the output of a sensor, and leakage in pipelines are examples of events that can lead to failures. Failure diagnosis is the process of detecting and identifying such deviations in a system using the information available through sensors. The problem of failure diagnosis has received considerable attention in the literature of reliability engineering, control, and computer science; and a wide variety of schemes have been proposed. Recently, it has also been studied in the framework of discrete event systems (DESs) [5, 6, 7, 13, 79, 38, 22, 77, 46, 59, 64, 65, 98, 99, 96, 23, 112, 117, 118, 93, 71, 31].

A notion of failure diagnosis of qualitative behaviors of discrete event systems was first proposed in [98]. The idea is that if the DES executes a faulty event, then it must be diagnosed within a bounded number of state-transitions/events. A method for constructing a diagnoser was developed, and a necessary and sufficient condition of diagnosability was obtained in terms of certain properties of the constructed diagnoser. The above work was

further extended to timed systems in [13] and to decentralized diagnosis in [23]. In [46], an algorithm of polynomial complexity for testing diagnosability without having to construct a diagnoser was obtained. This later work enabled a quick test for diagnosability; by applying this test a diagnoser is constructed only for those systems that are diagnosable. Note that the off-line construction of a diagnoser is of exponential complexity [98].

In [64, 65], the authors proposed a state-based approach for diagnosis; they studied the problems of off-line and on-line diagnosis where the basic idea was to "test and observe". Extensions of the above work can be found in [5] where the authors studied testability of DESs. In [6, 7], the problem of failure detection in communication networks was studied, where both the normal and faulty behaviors of the system were modeled by formal languages. In [79], the authors also studied the problem of fault detection in communication networks where faults are specified as change and addition of arcs in the finite state machine model of the normal system, and a diagnosis method was provided. In [93], a state-based approach for failure diagnosis of timed systems was proposed. In [38, 22, 77], the authors developed a template based monitoring scheme using timing and sequencing relationships of events for fault monitoring in manufacturing systems. In [112], the application of DESs techniques to digital circuits was studied, and an algorithm for the delay fault testability modeling and analysis was presented.

In most above works, the non-faulty behavior of the system, also called the specification, is either specified by an automaton (containing no failure states) or by a language (event-traces containing no failure events). Since in practical setting, a specification is generally given in a natural language, we need to first transform a natural language specification into a formal language specification before we apply the above failure diagnosis results. Given a simple natural language specification, the process of finding a corresponding formal language specification can be tedious, unintuitive, and error-prone, making it unaccessible to non-specialists. So there exists a gap between the informal natural language specification and the corresponding formal language specification. Temporal logic based specification was proposed in [26] as an attempt to bridge such a gap. Temporal logic has been used in the analysis and control of DESs [48, 103, 102, 115, 80, 4, 52, 75, 76, 68, 105, 63, 69]; and it has also been used as a formalism for diagnosing DESs in [84, 47, 20].

In this chapter, we study the failure diagnosis problem for systems modeled in a rules based model [12], extended to include faults [43]. State variables and rules for modifying their values are used to compactly model a DES. The representation of a system with

faults, in the rules based modeling formalism, is polynomial in the size of signals and faults. The compactness of this model, together with its intuitive nature, makes it user-friendly, less error-prone, more flexible, easily scalable, and provides canonicity of representation for models of systems with faults. The motivation of the work presented here is to develop techniques for failure diagnosis that are able to exploit the compactness of the model. In this regard, we develop techniques based on 1st-order temporal logic model-checking and predicates and predicate transformers.

The rules based modeling formalism is based on an *input/output* view of the system. The input signals of the system are the independent variables, and output signals the dependent variables which are a function of the independent variables and of other dependent variables. For simplicity, all signals in the system are assumed to be binary valued (extension to non-binary valued signals has been considered in rule-based formalism [12]), and it is also assumed that the state of the system can be specified by the current values of the signals (extension to the case when the state depends on also the past values of the signals has also been considered in [12]). In order to model failure prone systems, a binary valued fault signal is introduced to model presence or absence of each fault. From a modeling standpoint, the fault signals are treated just the same as any other signal in the system. Addition of fault signals to capture the faulty behavior requires new rules for the added fault events, and modification of rules of existing non-faulty events, by appropriately weakening their enabling guard conditions.

In the rule-based model, *initial conditions* are used to specify the initial values of the system signals. An event is a transition of an input or an output signal from one binary value to another. For each of the input and output events of the system (which includes the fault events), we obtain *event occurrence rules*. The antecedent of such a rule is a predicate over the signal values that serves as an enabling condition.

In this chapter we use 1^{st} order model checking for testing diagnosability of DESs, and predicates and predicate transformers for building an online diagnoser. We illustrate through various examples how the diagnosability of DESs modeled using a rules based formalism [12] prone to faults can be checked, and how an online diagnoser for the system can be constructed.

The rest of the chapter is organized as follows. In Section 2, the definitions of predicates and predicate transformers, rules based model, diagnosability, and 1^{st} order LTL temporal logic are introduced. In Section 3, diagnosability as a 1^{st} order LTL temporal logic model-checking is studied, and illustrated via an example. An algorithm for on-line diagnoser is

provided in Section 4 and illustrated using an example. Conclusions is provided in Section 5.

5.2 Notation and Preliminaries

5.2.1 Predicates, their Transformers, and Rule-based Model

A discrete event system, denoted G, is a 4-tuple $G := (X, \Sigma, \rightsquigarrow, X_0)$, where X denotes the state set, Σ is the finite event set, $\rightsquigarrow \subseteq X \times \Sigma \times X$ is the set of state transitions, and $X_0 \subseteq X$ is the set of initial states. We use state variables to represent the states and a finite set of conditional assignment statements, called rules, to represent the state transitions.

The notation $\vec{v}$ is used to denote the vector of state variables of G. If $\vec{v}$ is n-dimensional, then $\vec{v} = [v_1, \ldots, v_i, \ldots, v_n]$, where v_i is the ith state variable. The state space X of G equals the Cartesian product of domains of all state variables, i.e., $X := \prod_{i=1}^{n} \mathcal{D}(v_i)$, where $\mathcal{D}(v_i)$ is the domain of v_i. By definition $\mathcal{D}(v_i)$ is a countable set and can be identified with the set of natural numbers $\mathcal{N}$.

We use *predicates* for describing various subsets of the state space. Let $\mathcal{P}(\vec{v})$ denote the collection of predicates defined using the state variable vector $\vec{v}$, i.e., if $P(\vec{v}) \in \mathcal{P}(\vec{v})$, then it is a boolean valued map $P(\vec{v}) : X \to \{0, 1\}$. Consider for example a two dimensional state space $X = \mathcal{Z}^2$. Then the predicate $P(\vec{v}) = [v_1 \geq v_2]$ refers to all the states in which the value of variable v_1 is at least as large as the value of variable v_2. The symbols *true* and $false$ are used for denoting predicates that hold on all and none of the states respectively. With every predicate $P(\vec{v}) \in \mathcal{P}(\vec{v})$, we associate a set $X_P \subseteq X$ on which $P(\vec{v})$ takes the value one. Thus the collection of predicates $\mathcal{P}(\vec{v})$ has a one-to-one correspondence with the power set 2^X, and the names predicates and state-sets can be used interchangeably. We say that the predicate $P(\vec{v})$ holds on $\hat{X} \subseteq X$ if $\hat{X} \subseteq X_P$.

State transitions map a state to another state. Such mappings are extended to set of states or predicates in a natural way, and are known as *predicate transformers*. We use $\mathcal{F}$ to denote the collection of all predicate transformers, i.e., if $f \in \mathcal{F}$, then $f : \mathcal{P}(\vec{v}) \to \mathcal{P}(\vec{v})$. The *conjunctive closure* of f, denoted f_*, and *disjunctive closure* of f denoted f^* is defined to be $\bigwedge_{i \geq 0} f^i$ and $\bigvee_{i \geq 0} f^i$ respectively, where f^0 is the identity predicate transformer and $f^{i+1} := f(f^i)$. Given $f : X \to X$, the *substitution predicate transformer* "$\vec{v} \rightsquigarrow f(\vec{v})$" maps a

predicate $P(\vec{v})$ to $P(f(\vec{v}))$. Consider for example the f given by

$$(v_1, v_2) \rightsquigarrow (v_1 + v_2, v_1 - v_2).$$

Then the corresponding substitution predicate transformer maps the predicate $[v_1 < v_2]$ to $[v_1 + v_2 < v_1 - v_2] = [v_2 < 0]$.

Next we review the rule-based model [12] (which is a specific assignment program model [56]) for representing a DES G described above. The initial state set of G is specified as an *initial predicate*, denoted $I(\vec{v})$, which implies $X_0 = X_I$. The state transitions "$\rightsquigarrow$" of G is specified using a finite set of rules, also called conditional assignment statements, of the form:

$$\sigma : [C_\sigma(\vec{v})] \Rightarrow [\vec{v} \rightsquigarrow f_\sigma(\vec{v})],$$

where $\sigma \in \Sigma$ is an event, $C_\sigma(\vec{v})$ is a predicate, called the enabling condition or the *guard*, and $f_\sigma : X \to X$ is a map defined on the state space. If no guard is present, then *true* is treated as the guard. A conditional assignment statement of the above type is *enabled* if the condition $C_\sigma(\vec{v})$ holds. An enabled assignment statement may *execute*. Upon execution, new values are assigned to the state variables according to the map f_σ and a state transition on the event σ occurs. For simplicity, we assume that if multiple assignment statements are simultaneously enabled, only one of them is nondeterministically executed. This assumption may be relaxed to allow *concurrency* of execution.

The following example illustrates the representation of a DES using the rules based modeling formalism.

Example 2 Consider a tank filling system whose schematic is shown in Figure 5.1. It has

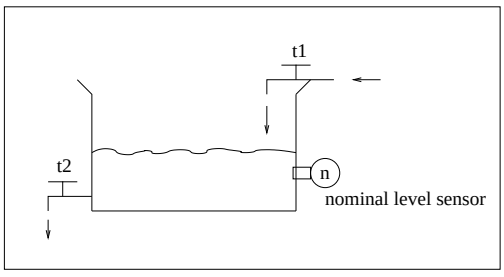

Figure 5.1: Tank system schematic

one filling tap *t1*, one draining tap *t2*, and a nominal level sensor *n*. The signals in the system are *t1, t2, n*, and the events that can occur in this system are *t1on, t1off, t2on, t2off,*

nup, ndn. Assume that the filling rate of tap *t1* has the flow value of +10, while that of the draining is +1.

The model in the rules based formalism, consisting of initial conditions and event occurrence rules, is given in Figure 5.2. Note that the rule for *nup* simply states that for the event *nup* to occur, the system should be in a state where the filling rate exceeds the draining rate, and the level sensor is down. The rule for *ndn* is similar, and other rules are default ones.

1. Initial conditions: $t1, t2, n = [\texttt{off}, \texttt{off}, \texttt{down}]$.

2. Event occurrence rules:

$$nup : [(t1^{+10} > t2^{+1}) \wedge \overline{n}] \Rightarrow [\overline{n} \rightsquigarrow n]$$
$$ndn : [(t2^{+1} > t1^{+10}) \wedge n] \Rightarrow [n \rightsquigarrow \overline{n}]$$
$$t1on : [\overline{t1}] \Rightarrow [\overline{t1} \rightsquigarrow t1]; \qquad t1off : [t1] \Rightarrow [t1 \rightsquigarrow \overline{t1}]$$
$$t2on : [\overline{t2}] \Rightarrow [\overline{t2} \rightsquigarrow [t2]; \qquad t2off : [t2] \Rightarrow [t2 \rightsquigarrow \overline{t2}]$$

Figure 5.2: Rules based model of the tank system

The substitution predicate transformer can be used to define the *forward one-step reachable*, fr, and *backward one-step reachable*, br, predicate transformers for G. fr determines the "postcondition" after the occurrence of a state transition for a given "precondition", whereas br determines the "precondition" prior to the occurrence of a state transition for a given postcondition.

For the assignment statement $\sigma : [C_\sigma(\vec{v})] \Rightarrow [\vec{v} \rightsquigarrow f_\sigma(\vec{v})]$ and a condition $P(\vec{v})$, these are formally defined as follows:

$$fr(P(\vec{v}), \sigma) := C_\sigma(f_\sigma^{-1}(\vec{v})) \wedge P(f_\sigma^{-1}(\vec{v})); \quad br(P(\vec{v}), \sigma) := C_\sigma(\vec{v}) \wedge P(f_\sigma(\vec{v})).$$

Note that the computation of br is easier as compared to that of fr, since its computation does not require the extra computation of f^{-1}.

For $\hat{\Sigma} \subseteq \Sigma$, we define $fr(P(\vec{v}), \hat{\Sigma}) := \bigvee_{\sigma \in \hat{\Sigma}} fr(P(\vec{v}), \sigma)$, and similarly, $br(P(\vec{v}), \hat{\Sigma}) := \bigvee_{\sigma \in \hat{\Sigma}} br(P(\vec{v}), \sigma)$. Finally, note that $fr^*(P(\vec{v}), \hat{\Sigma})$ denotes the set of states which are reachable from a state in $P(\vec{v})$ by execution of zero or more transitions of events in $\hat{\Sigma}$. Similarly, $br^*(P(\vec{v}), \hat{\Sigma})$ denotes the set of states from where a state in $P(\vec{v})$ can be reached by execution

of zero or more transitions of events in $\hat{\Sigma}$. Clearly, fr^* is useful in characterizing the *forward reachability*, whereas br^* is useful in characterizing the *backward reachability*.

Given $f \in \mathcal{F}$ and $P(\vec{v}) \in \mathcal{P}(\vec{v})$, the *restriction* of f to $P(\vec{v})$, denoted $f \mid P(\vec{v})$, is the predicate transformer defined as:

$$f \mid P(\vec{v})(Q(\vec{v})) := f(P(\vec{v}) \bigwedge Q(\vec{v})) \bigwedge P(\vec{v}), \forall Q(\vec{v}) \in \mathcal{P}(\vec{v}).$$

5.2.2 1^{st} order LTL temporal logic & model checking

Propositional linear temporal logic (PLTL) [26] is an extension of propositional logic (PL) by the *temporal logic quantifiers/operator* $\{X, U, F, G, B\}$, called next time, until, eventually, always, and before, respectively. The temporal operators describe properties of a state-trace in a computation:

- X ("next time"): it requires that a property hold in the next state of the state-trace.
- U ("until"): it is used to combine two properties. The combined property holds if there is a state in the state-trace where the second property holds, and at every preceding state in the trace, the first property holds.
- F ("eventually" or "in the future"): it is used to assert that a property will hold at some future state in the state-trace. It is a special case of "until".
- G ("always" or "globally"): it specifies that a property holds at every state in the state-trace.
- B ("before"): it also combines two properties. It requires that if there is a state in the state-trace where the second property holds, then there exists a preceding state in the trace where the first property holds.

We have following relations among the above operators, where f denotes a temporal logic formula:

- $Ff \equiv TrueUf$
- $Gf \equiv \neg F \neg f$
- $fBg \equiv \neg(\neg fUg)$

So X and U can be used to express the other temporal operators. These are the only temporal operators that appear in the definition of linear temporal logic.

The following examples show that PLTL temporal logic formulae can be used to easily express properties such as invariance, recurrence, stability, etc.

Gp means that "along a given state-trace, *globally* (G) at every state of the trace, p is true". It is an invariance (a type of safety) property.

$G(p_1 \Rightarrow Fp_2)$ means that "along a given state-trace, *globally* (G) for every state s of the trace, if p_1 is true at the state s, then p_2 will be true at some *future* (F) state". It is a recurrence (a type of liveness) property.

FGp means that "along a given state-trace, *eventually* (F) p will hold *globally* (G)". It is a property of stability (a type of liveness) which requires that the system should eventually reach a set of states where p holds and stay there forever.

First-order linear temporal logic (FOLTL) [26] is obtained by taking propositional linear temporal logic and adding to it a First order language **L**. That is, in addition to atomic propositions, truth-function connectives, and temporal operators we now also have predicates, functions, individual constants, and individual variables, each interpreted over appropriate domain.

A first order language **L** consists of *variable* symbols, *function* symbols and a set of *predicate* symbols. The zero-ary function symbols comprise the subset of *constant* symbols. Similarly, the zero-ary predicate symbols are known as the *proposition* symbols. We also have the predicate equality symbol $\approx$, and the quantifier symbols $\forall$ and $\exists$, which are applied to individual variable symbols, using the usual rules regarding scope of quantifiers, and free and bound variables.

The *term* of **L** are defined inductively by the following rules:

T1 Each constant c is a term.

T2 Each variable y is a term.

T3 If $t_1, \cdots, t_n$ are terms and f is an n-ary function symbol then $f(t_1, \cdots, t_n)$ is a term.

The *atomic formulae* of **L** are defined by the following rules:

AF1 Each 0-ary predicate symbol (i.e. atomic proposition) is an atomic formula.

AF2 If $t_1, \cdots, t_n$ are terms and ψ is an n-ary predicate then $\psi(t_1, \cdots, t_n)$ is an atomic formula.

AF3 If t_1, t_2 are terms then $t_1 \approx t_2$ is also an atomic formula.

Finally, the compound formulae of **L** are defined inductively as follows:

F1 Each atomic formula is a formula.

F2 If p, q are formulae then $(p \wedge q)$, $\neg p$ are formulae.

F3 If p is a formula and y is a free variable in p then $\exists y\ p$ is a formula.

The semantics of **L** is provided by an interpretation I over some domain D. The interpretation I assigns an appropriate meaning over D to the (non-logic) symbols of **L**: Essentially, the n-ary predicate symbols are interpreted as concrete, n-ary relations over D, while the n-ary function symbols are interpreted as concrete, n-ary functions on D.

For defining FOLTL, we assume that the set of symbols is divided into two classes, the class of *global* symbols and the class of *local* symbols. Intuitively, each global symbol has the same interpretation over all states; the interpretation of local symbol may vary, depending on the state at which it is evaluated. We now define the language of FOLTL obtained by adding **L** to PLTL. First, the terms of FOLTL are those generated by rules $T1 - 3$ for **L** plus the rule:

T4 If t is a term, then Xt is a term (intuitively, denoting the immediate future value of term t).

Finally, the compound formulae of FOLTL are defined inductively using the following rules:

FOLTL1 Each atomic formula is a formula.

FOLTL2 If p, q are formulae, then so are $p \wedge q$, $\neg p$.

FOLTL3 If p, q are formulae, then so are PUq, Xp.

FOLTL4 If p is a formula and y is a free variable in p, then $\exists y\ p$ is a formula.

The semantics of FOLTL is provided by a first order linear time structure $M = (S, x, L)$, where S is state set, $x : \mathcal{N} \rightarrow S$ is an infinite state sequence, and L associates with each

state s an interpretation $L(s)$ of all symbols at s over a domain D such that, for each global symbol w, $L(s)(w) = L(s')(w)$, for all $s, s' \in S$.

Since the terms of FOLTL are generated by rules $T1-3$ for **L** plus the rule $T4$ above, we extend the meaning function - denoted by a pair (M, x) – for terms:

$(M, x)(c) = L(\cdot)(c)$, since all constants are global.

$(M, x)(y) = L(\cdot)(y)$, where y is a global variable.

$(M, x)(y) = L(s_0)(y)$, where y is a local variable and $x = (s_0, s_1, s_2, \cdots)$.

$(M, x)(f(t_1, \cdots, t_n)) = (M, x)(f)((M, x)(t_1), \cdots, (M, x)(t_n))$.

$(M, x)(Xt) = (M, x^1)(t)$.

Now the extension of $\models$ is routine. For atomic formulae we have:

$M, x \models P$ iff $L(\cdot)(P) = true$, where P is a global proposition.

$M, x \models P$ iff $L(s_0)(P) = true$, where P is a local proposition and $x = (s_0, s_1, s_2, \cdots)$.

$M, x \models \psi(t_1, \cdots, t_n)$ iff $(M, x)(\psi)((M, x)(t_1), \cdots, (M, x)(t_n)) = true$.

$M, x \models t_1 \approx t_2$ iff $(M, x)(t_1) = (M, x)(t_2)$.

We finish off the semantics of FOLTL with-the inductive definition of $\models$ for compound formulae:

$M, x \models p \wedge q$ iff $M, x \models p$ and $M, x \models q$.

$M, x \models \neg p$ iff it is not the case that $M, x \models p$.

$M, x \models (pUq)$ iff $\exists j(M, x^j \models q$ and $\forall k < j(M, x^k \models p))$.

$M, x \models Xp$ iff $M, x^1 \models p$.

$M, x \models \exists y\ p$, where y is global variable free in p, iff there exists some $d \in D$ for which $M[y \leftarrow d], x \models p$, where $M[y \leftarrow d]$ is the structure having global interpretation $I[y \leftarrow d]$ identical to I except y is assigned the value d.

A formula p of FOLTL is *valid* iff for every first order linear time structure $M = (S, x, L)$ we have $M, x \models p$. The formula p is *satisfiable* iff there exists $M = (S, x, L)$ such that $M, x \models p$.

5.3 Diagnosability as 1^{st} order LTL model-checking

In order to test the diagnosability in the rule-based model, we adopt the test for diagnosability in the automaton setting presented in our past work [46]. The test in the automaton setting consisted of the following steps:

- Refine the state set X of G by augmenting each state by a binary value label such that for each $x \in X$, $(x, 1)$ (resp., $(x, 0)$) represents the traces in $L(G)$ that lead to x and contain a (resp., no) failure event.
- Take synchronous composition of two copies of the refined G, by first replacing each event label σ by $M(\sigma)$. This is called *masked synchronous composition.*
- Check if the synchronous composition contains a cycle of state-pairs where the two components carry non-identical labels. (G is diagnosable if and only if no such cycles are found.)

As in the automaton setting, we introduce a binary valued variable F to indicate whether or not a fault happened in past. with this the new state variable set becomes,

$$\vec{x} := (\vec{v}, F).$$

We next need to extend the rule-based model to include this new state-variable. Assuming that the system starts in a non-faulty state, the initial state is given by the predicate,

$$I(\vec{x}) := I(\vec{v}) \wedge [F = 0].$$

The rule for each event $\sigma \in \Sigma$, $[C_\sigma(\vec{v})] \Rightarrow [\vec{v} \rightsquigarrow f_\sigma(\vec{v})]$ is extended as follows. For a non-faulty event,

$$[C_\sigma(\vec{v})] \Rightarrow [(\vec{v}, F) \rightsquigarrow (f_\sigma(\vec{v}), F)$$

(non-faulty event retains the value of F as unchanged), and for a faulty event,

$$[C_\sigma(\vec{v})] \Rightarrow [(\vec{v}, F) \rightsquigarrow (f_\sigma(\vec{v}), 1)$$

(faulty event makes the value of F equal to 1).

To facilitate diagnosis, we define a faulty-state predicate,

$$B(\vec{x}) = B((\vec{v}, F)) := [F = 1].$$

Using this predicate and the extended rule-base model (which includes the new boolean variable F, new initial condition, and new assignment statements), we perform the diagnosis test as follows.

Algorithm 2 Consider G with state variables $\vec{v}$, event set Σ, and model given by:

Initial condition: $I(\vec{v}) \in \mathcal{P}(\vec{v})$, and
Event occurrence rules: $\forall \sigma \in \Sigma : [C_\sigma(\vec{v})] \Rightarrow [\vec{v} \rightsquigarrow f_\sigma(\vec{v})]$.

The set of fault events is denoted by $\Sigma_F \subseteq \Sigma$, and the events are partially observed through a event observation mask $M : \Sigma \cup \{\epsilon\} \to \Delta \cup \{\epsilon\}$ with $M(\epsilon) = \epsilon$, and $M(\sigma) = \epsilon$ for each $\sigma \in \Sigma_F$.

- Augment the state variables by a boolean variable, F, to identify whether or not a fault happened in past. The augmented state variable is given by, $\vec{x} = (\vec{v}, F)$. The augmented system model is given by,

  Initial condition: $I(\vec{x}) = I(\vec{v}) \wedge [F = 0]$
  Event occurrence rules:

  $$\forall \sigma \in \Sigma_F : [C_\sigma(\vec{v})] \Rightarrow [\vec{x} \rightsquigarrow (f_\sigma(\vec{v}), 1)]$$
  $$\forall \sigma \notin \Sigma_F : [C_\sigma(\vec{v})] \Rightarrow [\vec{x} \rightsquigarrow (f_\sigma(\vec{v}), F)]$$

  Denote the set of states that are visited after a fault has happened in past by the predicate, $B(\vec{x}) = B((\vec{v}, F)) := [F = 1]$.

- Perform a "masked synchronous composition" of augmented G with itself to obtain the system G_d (here $\vec{x}$ and $\vec{y}$ are used to denote the state-variables of the two copies of the augmented G):

  Initial condition: $I(\vec{x}) \bigwedge I(\vec{y})$.
  Event occurrence rule: $\forall (\sigma, \sigma') \in [(\Sigma \cup \{\epsilon\})^2 - \{\epsilon, \epsilon\}]$ s.t. $M(\sigma) = M(\sigma')$:

  $$\begin{array}{rcll} [C_{M(\sigma)}(\vec{x}) \bigwedge C_{M(\sigma')}(\vec{y})] & \Rightarrow & [(\vec{x}, \vec{y}) \rightsquigarrow (f_{M(\sigma)}(\vec{x}), f_{M(\sigma')}(\vec{y}))] & \text{if } \sigma, \sigma' \neq \epsilon \\ [C_{M(\sigma)}(\vec{x})] & \Rightarrow & [(\vec{x}, \vec{y}) \rightsquigarrow (f_{M(\sigma)}(\vec{x}), \vec{y})] & \text{if } \sigma \neq \epsilon, \sigma' = \epsilon \\ [C_{M(\sigma')}(\vec{y})] & \Rightarrow & [(\vec{x}, \vec{y}) \rightsquigarrow (\vec{x}, f_{M(\sigma')}(\vec{y}))] & \text{if } \sigma = \epsilon, \sigma' \neq \epsilon \end{array}$$

- Using 1^{st} order linear-time temporal logic model checking check whether there exists an "ambiguous" cycle by model-checking the following formula in G_d:

$$\begin{aligned} & \exists \vec{x_0}, \vec{y_0}[EGF(\vec{x} = \vec{x_0} \bigwedge \vec{y} = \vec{y_0} \bigwedge B(\vec{x_0}) \bigwedge \neg B(\vec{y_0}))] \\ \equiv \quad & \exists \vec{x_0}, \vec{y_0}[EGAF(\vec{x} = \vec{x_0} \bigwedge \vec{y} = \vec{y_0} \bigwedge B(\vec{x_0}) \bigwedge \neg B(\vec{y_0}))]. \end{aligned}$$

  Then G is diagnosable if and only if the above formula does not hold in G_d.

The formula to be model-checked checks the for existence of a state pair $(\vec{x}_0, \vec{y}_0) \in (X \times \{0,1\})^2$ with the property that

- $\vec{x}_0$ is a "faulty" state: $B(\vec{x}_0)$ holds,
- $\vec{y}_0$ is a "non-faulty" state: $\neg B(\vec{y}_0)$ holds,
- $(\vec{x}_0, \vec{y}_0)$ is visited infinitely often along some state trajectory starting from the initial condition $I(\vec{x}) \wedge I(\vec{y})$: $EGF\vec{x} = \vec{x}_0 \wedge \vec{y} = \vec{y}_0$, i.e., exists a path (E) such that globally (G) along each state of the path, in future (F) it holds that $\vec{x} = \vec{x}_0$ and $\vec{y} = \vec{y}_0$.

Whenever the above formula is satisfiable, there exists a pair of faulty and non-faulty traces in G of arbitrary long length that are indistinguishable, and the system is not diagnosable. The model-checking software tools such *NuSMV* [15] can be used to check the satisfiability of the above formula in G_d.

Example 3 In order to illustrate our result, we give a simple example which consists of a traffic monitoring problem of a mouse in a maze. The maze, shown in Figure 5.3, consists of four rooms connected by various one-way passages, where some of them have sensors installed to detect the passing of the mouse. There is also a cat which alway stays in room 1. The mouse is initially in room 0, and it can visit other rooms by using one way passages, and it never stays at one room forever. A failure occurs when the mouse visits the room occupied by the cat. Our task is to monitor the behavior of the mouse by observing the sensor signals to detect whether or not a failure occurred.

The above problem can be formulated as a failure diagnosis problem in the rules based model setting. The system to be diagnosed, G, has a single state variable v denoting the location of the mouse in the maze, and it can take the values in the set $\{0,1,2,3\}$; the initial state is $I(v) = [v = 0]$; the event set is $\Sigma = \{o_1, o_2, o_3, u_1, u_2, u_3\}$; the event observation mask M is given as $M(u_i) = \epsilon$ and $M(o_i) = o_i$ for $1 \leq i \leq 3$. The rules based model of mouse in a

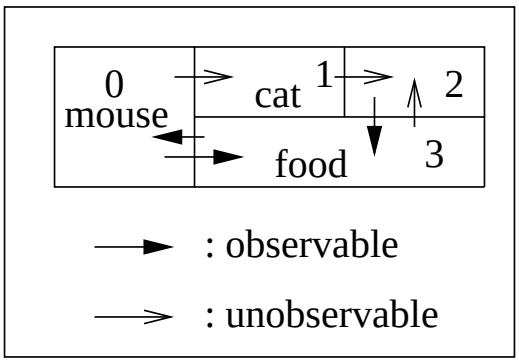

Figure 5.3: Mouse in a maze

1. Initial condition: $I(v) = [v = 0]$.

2. Event occurrence rules:

$$o_1 : [v = 0] \Rightarrow [v \rightsquigarrow 3]$$
$$o_2 : [v = 3] \Rightarrow [v \rightsquigarrow 0]$$
$$o_3 : [v = 2] \Rightarrow [v \rightsquigarrow 3]$$
$$u_1 : [v = 0] \Rightarrow [v \rightsquigarrow 1]$$
$$u_2 : [v = 1] \Rightarrow [v \rightsquigarrow 2]$$
$$u_3 : [v = 3] \Rightarrow [v \rightsquigarrow 2]$$

Figure 5.4: Rules based model of mouse in a maze

maze is shown in Figure 5.4. Since a fault occurs when room 1 is visited, $\Sigma_F = \{u_1\}$, where note that u_1 is an unobservable event.

In order to verify diagnosability, we augment the state variable v by the binary valued variable F to obtain $\vec{x} := (v, F)$. The augmented system model is shown in Figure 5.5.

1. Refined initial condition: $I(\vec{x}) = (I(v), 0) = [0, 0]$.

2. Event occurrence rules:

$$o_1 : [v = 0] \Rightarrow [\vec{x} \rightsquigarrow (3, F)]$$
$$o_2 : [v = 3] \Rightarrow [\vec{x} \rightsquigarrow (0, F)]$$
$$o_3 : [v = 2] \Rightarrow [\vec{x} \rightsquigarrow (3, F)]$$
$$u_1 : [v = 0] \Rightarrow [\vec{x} \rightsquigarrow (1, 1)]$$
$$u_2 : [v = 1] \Rightarrow [\vec{x} \rightsquigarrow (2, F)]$$
$$u_3 : [v = 3] \Rightarrow [\vec{x} \rightsquigarrow (2, F)]$$

Figure 5.5: Augmented rules based model of mouse in a maze

Next using the state variable $\vec{y} = (u, E)$ for the second copy of G, we compute the masked composition of augmented G with itself to obtain G_d as shown in Figure 5.6. Note that the observable events o_1, o_2, o_3 execute synchronously, whereas the unobservable events u_1, u_2, u_3 occur asynchronously.

We used the NuSMV tool [15] for model-checking the diagnosability condition:

$$\exists \vec{x_0}, \vec{y_0}[EGAF(\vec{x} = \vec{x_0} \bigwedge \vec{y} = \vec{y_0} \bigwedge B(\vec{x_0}) \bigwedge \neg B(\vec{y_0}))].$$

This NuSMV tool allows computation of masked synchronous composition. We verified that the mouse in a maze is diagnosable (as expected from our automaton based computation). Since a rules based model provides a compact model (in contrast, an automaton model enumerates all the states), we hope that the symbolic techniques developed in this chapter will allow for the diagnosability verification of industrial size problems.

Further, we can use the diagnosability algorithm developed above together with the optimal sensor selection algorithm given in [50] to obtain an optimal observation mask while preserving the system diagnosability. Using this approach, we determined that the event o_3 need not be observable for the system to remain diagnosable.

1. Initial condition: $I(\vec{x},\vec{y}) = (I(v), 0, I(u), 0) = [0,0,0,0]$.

2. Event occurrence rules:

$$
\begin{aligned}
&(o_1, o_1) : [v = 0] \wedge [u = 0] \Rightarrow [(\vec{x},\vec{y}) \rightsquigarrow (3, F, 3, F)]\\
&(o_2, o_2) : [v = 3] \wedge [u = 3] \Rightarrow [(\vec{x},\vec{y}) \rightsquigarrow (0, F, 0, F)]\\
&(o_3, o_3) : [v = 2] \wedge [u = 2] \Rightarrow [(\vec{x},\vec{y}) \rightsquigarrow (3, F, 3, F)]\\
&(u_1, \epsilon) : [v = 0] \Rightarrow [(\vec{x},\vec{y}) \rightsquigarrow (1, 1, \vec{y})]\\
&(\epsilon, u_1) : [u = 0] \Rightarrow [(\vec{x},\vec{y}) \rightsquigarrow (\vec{x}, 1, 1)]\\
&(u_2, \epsilon) : [v = 1] \Rightarrow [(\vec{x},\vec{y}) \rightsquigarrow (2, F, \vec{y})]\\
&(\epsilon, u_2) : [u = 1] \Rightarrow [(\vec{x},\vec{y}) \rightsquigarrow (\vec{x}, 2, F)]\\
&(u_3, \epsilon) : [v = 3] \Rightarrow [(\vec{x},\vec{y}) \rightsquigarrow (2, F, \vec{y})]\\
&(\epsilon, u_3) : [u = 3] \Rightarrow [(\vec{x},\vec{y}) \rightsquigarrow (\vec{x}, 2, F)]
\end{aligned}
$$

Figure 5.6: Masked synchronous composition of two augmented mouse in a maze

5.4 On-line Diagnoser using Predicates & their Transformers

We embark upon the on-line computation of a diagnoser once the system has been determined to be diagnosable. Again as with the test for diagnosability, we develop symbolic methods for the on-line computation of the diagnoser.

The diagnoser maintains two predicates: one, denoted $E_k(\vec{x}) \in \mathcal{P}(\vec{x})$, is an estimate of the possible states following the occurrence of kth observable event, and the other denoted $N_k(\vec{x}) \in \mathcal{P}(\vec{x})$ is a subset of $E_k(\vec{x})$ that is reached along trajectories that never visit a state in $B(\vec{x})$, i.e., along those non-faulty trajectories where $\neg B(\vec{x})$ holds invariantly.

Initially, when no observation has occurred, i.e., when $k = 0$,

$$E_0(\vec{x}) = I(\vec{x}), \quad N_0(\vec{x}) = I(\vec{x}) \wedge \neg B(\vec{x}).$$

Upon the occurrence of the $(k+1)$th observable event ($k \geq 0$), the pair $(E_k(\vec{x}), N_k(\vec{x}))$ is updated to obtain the pair $(E_{k+1}(\vec{x}), N_{k+1}(\vec{x}))$. Whenever $N_k(\vec{x})$ is a strict subset of $E_k(\vec{x})$, and $N_k(\vec{x}) \neq false$, it means that the system could have executed some trajectories that visited a faulty state in past (since $E_k(\vec{x}) \neq false$), and also some other trajectories

(that are indistinguishable to the former) that never visited a faulty state in past (since $N_k(\vec{x}) \neq false$). In other words, in such a case, there exists an ambiguity as to whether or not a fault occurred in past. Such an ambiguity does not exist if

$$[E_k(\vec{x}) \neq false] \wedge [N_k(\vec{x}) = false],$$

which means that along all trajectories the system could have executed, a faulty state was visited in past. A fault is reported by the diagnoser at such a point.

Algorithm 3

- **Initiation step:**

$$E_0(\vec{x}) = I(\vec{x})$$
$$N_0(\vec{x}) = I(\vec{x}) \bigwedge \neg B(\vec{x})$$

- **Iteration step:** Upon $(k+1)$th observation $\delta \in M(\Sigma) - \{\epsilon\}$:

$$E_{k+1}(\vec{x}) = fr_{M^{-1}(\delta)}[fr^*_{M^{-1}(\epsilon)\cap\Sigma}(E_k(\vec{x}))]$$
$$N_{k+1}(\vec{x}) = (fr|_{\neg B(\vec{x})})_{M^{-1}(\delta)}[(fr|_{\neg B(\vec{x})})^*_{M^{-1}(\epsilon)\cap\Sigma}(N_k(\vec{x}))]$$

  Declare a fault if:

$$[E_{k+1}(\vec{x}) \neq false] \wedge [N_{k+1}(\vec{x}) = false].$$

In the iteration step, $E_{k+1}(\vec{x})$ is computed using a reachability computation starting from $E_k(\vec{x})$ on sequences of unobservable events in $M^{-1}(\epsilon) \cap \Sigma$ followed a single event in $M^{-1}(\delta)$ (since the $(k+1)$th observation of δ results from the execution of a sequence of unobservable events in $M^{-1}(\epsilon) \cap \Sigma$ followed by the execution of an event in $M^{-1}(\delta)$). $N_{k+1}(\vec{x})$ is computed in a similar way except the forward reachability predicate transformer fr is replaced by it's restriction to $\neg B(\vec{x})$, i.e., by $(fr|_{\neg B(\vec{x})})$.

We illustrate the above algorithm for on-line computation of the diagnoser using the example of mouse in a maze given earlier.

Example 4 Refer to the example of Section 5.3. We can compute the diagnoser as follows:

- $k = 0$**:** Since $I(\vec{x}) = [v = F = 0]$ and $B(\vec{x}) = [F = 1]$, we set

$$E_0(\vec{x}) = [v = F = 0], \quad N_0(\vec{x}) = [v = F = 0].$$

- $k = 1$**:** There are three observable events $o_1, o_2, o_3 \in \Sigma$. If the first observation is o_1, then

$$E_1(\vec{x}) = [v = 3, F = 0], \quad N_1(\vec{x}) = [v = 3, F = 0].$$

If the first observation is o_2, then

$$E_1(\vec{x}) = false, \quad N_1(\vec{x}) = false.$$

(This means o_1 as first observation is not possible.) If the first observation is o_3, then

$$E_1(\vec{x}) = [v = 3, F = 1], \quad N_1(\vec{x}) = false.$$

This means that a fault has occurred sometimes in past.

- $k = 2$**:** Suppose the first observation ($k = 1$) is o_1. If the next observation is o_1, then

$$E_2(\vec{x}) = false, \quad N_2(\vec{x}) = false.$$

(This means the observation sequence o_1o_1 is not possible.) If the next observation is o_2, then

$$E_2(\vec{x}) = [v = 0, F = 0] = E_0(\vec{x}), \quad N_2(\vec{x}) = [v = 0, F = 0] = N_0(\vec{x}).$$

If the next observation is o_3, then

$$E_2(\vec{x}) = [v = 3, F = 0], \quad N_2(\vec{x}) = [v = 3, F = 0].$$

Next suppose the first observation ($k = 1$) is o_2. Then since $E_1(\vec{x}) = N_1(\vec{x}) = false$, it follows that $E_2(\vec{x}) = N_2(\vec{x}) = false$.

Finally suppose the first observation ($k = 1$) is o_3. If the next observation is o_1, then

$$E_2(\vec{x}) = false, \quad N_2(\vec{x}) = false.$$

(This means the observation sequence o_3o_1 is not possible.) If the next observation is o_2, then

$$E_2(\vec{x}) = [v = 0, F = 1], \quad N_2(\vec{x}) = false.$$

If the next observation is o_3, then

$$E_2(\vec{x}) = [v = 3, F = 1], \quad N_2(\vec{x}) = false.$$

In both the above cases, we know that a fault has occurred in past.

- $k = 3$**:** After two observations the only predicate pair that is not "re-visited" is the one following the observation sequence o_3o_2:

$$E_2(\vec{x}) = [v = 0, F = 1], \quad N_2(\vec{x}) = false.$$

So we examine this predicate pair. If the next observation is o_1, then

$$E_3(\vec{x}) = [v = 3, F = 1], \quad N_3(\vec{x}) = false.$$

If the next observation is o_2, then

$$E_3(\vec{x}) = false, \quad N_3(\vec{x}) = false.$$

(This means the observation sequence $o_3o_2o_2$ is not possible.) If the next observation is o_3, then

$$E_3(\vec{x}) = [v = 3, F = 1], \quad N_3(\vec{x}) = false.$$

Thus the third iteration step does not introduce a predicate pair that has never been "visited" before, and so there is no need to iterate further. (Said another way, further iterations will yield a predicate pair that has already been computed above.)

In this case, the on-line computation of the three steps considered above yields a off-line diagnoser that can be represented as an automaton as shown in Figure 5.7. In Figure 5.7, when there is a transition from a predicate pair $[E(\vec{x}) \neq false] \wedge [N(\vec{x}) \neq false]$ to the predicate pair $[E(\vec{x}) \neq false] \wedge [N(\vec{x}) = false]$, a fault is reported by the diagnoser. Further since the predicate pair $[E(\vec{x}) = false] \wedge [N(\vec{x}) = false]$ represents an impossibility, the diagnoser can be reduced by restricting it to those predicate pairs that are of the type $[E(\vec{x}) \neq false] \wedge [N(\vec{x}) \neq false]$. This restricted diagnoser is shown in Figure 5.8. Any observation sequence that leads to being outside the restricted diagnoser automaton indicates that a fault must have occurred in past.

5.5 Conclusion

The rules based modeling formalism of [12] has been used to models discrete event systems prone to failures. Stuck-signal faults, and system/equipment faults can be easily modeled in the rules based modeling formalism. Symbolic computation based algorithms for checking the diagnosability of DESs using 1^{st} order model-checking and for online diagnoser synthesis

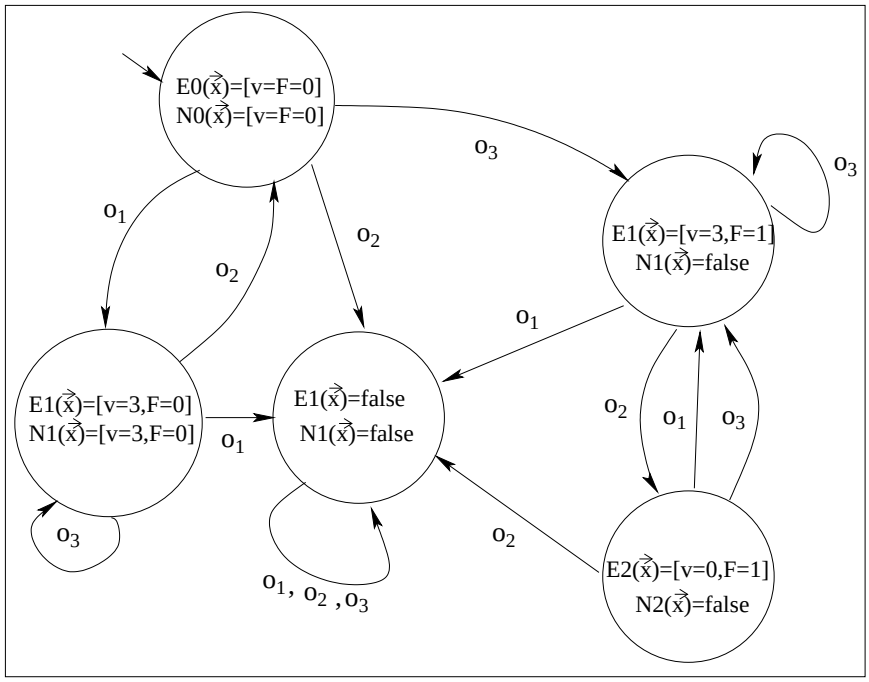

Figure 5.7: Diagnoser for mouse in a maze

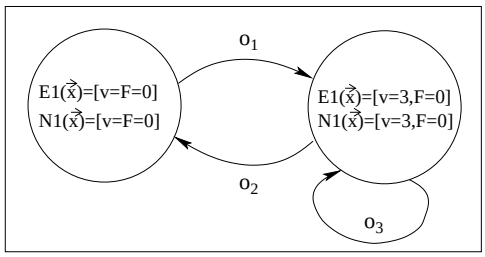

Figure 5.8: The reduced diagnoser for mouse in a maze

using predicates and predicate transformers have been developed. The advantage of using rule-based model is it's compactness since it uses state-variables to represent states. The number of rules in the rule-based model is polynomial in number of state variables. Symbolic methods allow for failure analysis without exhaustively performing a reachability of the entire state space. Plus, software tools such as NuSMV exist for performing 1^{st}-order model-checking for systems with finite/bounded state-space.

Chapter 6

Rules based Modeling of an Assembly Line and its Diagnosis

6.1 Introduction

In this chapter, we study the modeling and failure diagnosis of a miniature assembly line [11] in the rules-based model developed in [12], and later extended in [43] to also model faults. The present chapter demonstrates the applicability of the rules-based modeling and diagnosis techniques to practical manufacturing systems. The demonstration system is a simple educational test-bed built using LEGO® blocks that simulates an automated car assembly-line. This miniature assembly-line shown in Figure 2.7 performs the assembly of the roof and the chassis. These two parts are transported to the press section from their respective loading sections, where a vertical press operation presses the two parts together, and finally the assembled part exits the assembly-line through the unloading section. A transporter links the chassis, roof, press, and unloading sections. While setting up the miniature LEGO® assembly-line, the one built at the University of Massachusetts [10] served as a prototype.

We present the rules-based models of each of the individual sections, the composition of which is the entire plant model. The number of rules in transporter, chassis, roof, press, and unloading sections is 18, 12, 12, 10, and 6, respectively (compare this to a total of about 1.7×10^6 states for the entire assembly-line if they were to be modeled as an automaton). For demonstrating the 1^{st}-order temporal logic model-checking based diagnosis technique developed in [44], we consider a simplified model of the transporter section, and

analyze it's diagnosability properties. When the system under examination is not diagnosable, sensor refinement/augmentation can be used to make the system diagnosable. We illustrate through various examples drawn from the LEGO® assembly-line how sensor refinement/augmentation methods can be used to make the system diagnosable.

The rest of the chapter is organized as follows. A description and rules-based model of the LEGO® assembly-line is given in Section 2. Section 3 illustrates diagnosis in rules-based model using a simplified model of one section of the assembly-line, and also studies how to design diagnosable systems, again using examples from the LEGO® assembly-line. Finally conclusions are provided in Section 4.

6.2 Rule-based models for the Assembly-Line

Instead of having a single large model for the system, and for making modeling simpler, we develop smaller sized "sub-models" by partitioning the entire system into five sections, namely, transporter, chassis, roof, press, and unloading. Their description is preceded by a list of all the events possible in the LEGO® assembly-line which is given in Figure 6.1.

1. **Transporter:** Parts are transported from one assembly section to another via the transporter, which consists of a fixture that is connected to one end of a rack that is moved by a pinion powered from a gear box motor. An angle sensor mounted on the same shaft as that of the pinion, counts off the number of rotations of the axle through it, in order to determine the position of the fixture. The rules-based model of the transporter is given in Figure 6.2. The initial conditions of this section consist of the forward and reverse motor turned off ($\overline{Tf}, \overline{Tr}$), and the transporter positioned at the initial home/unloading position (*g, h, i, j, k, l*). When the forward motor is turned on, and the reverse motor is off, the transporter will leave (a) the home position, move to the press position (b), then leave (c) the press position and reach the roof position (d). It will then leave (e) the roof position and finally reach the chassis position (f).

   If we model the stuck-on fault for the forward motor (*Tf*), then the following two rules for the stuck-on fault (*TfsonF*) and stuck-on recovery (*TfsonR*) events will be added:

$$TfsonF : [Tf \wedge \overline{Tfson}] \Rightarrow [\overline{Tfson} \rightsquigarrow Tfson];$$
$$TfsoR : [Tfson] \Rightarrow [Tfson \rightsquigarrow \overline{Tfson}].$$

Section	Signals	Events	Controllable
Transporter	M1	Ifon/of: indexing slide motor on/off forward dirn. input events	Yes
	M1	Iron/of: indexing slide motor on/off reverse dirn. input events	Yes
	A5	a : Indexing slide leaving home station during forward movement output event	No
	A5	b/j : Indexing slide at press station during forward/reverse movement output events	No
	A5	c/k : Indexing slide leaving press station during forward/reverse movement o/p events	No
	A5	d/h : Indexing slide at roof station during forward/backward movement o/p events	No
	A5	g : Indexing slide leaving roof station during forward/reverse movement o/p event	No
	A5	f : Indexing slide at chassis station during reverse movement optput event	No
	A5	l : Indexing slide at home or unloading position during reverse movement o/p event	No
Chassis	M2	cCon/of: chassis conveyor motor on/off input events	Yes
	M3	pCon/of: chassis pusher motor on/off input events	Yes
	T2	pCup/dn: chassis pusher retracted/not−retracted output events	No
	L6	dCup/dn: part present/absent at chassis station dock output events	No
Roof	M4	cRon/of: roof conveyor motor on/off input events	Yes
	M5	pRon/of: roof pusher motor on/off input events	Yes
	T3	pRup/dn: roof pusher retracted/not−retracted output events	No
	L7	dRdn/up: part present/absent at roof station dock output events	No
Press	M6	pPfon/of: press pusher motor on/off input events	Yes
	M6	pPron/of: press pusher motor on/off in reverse dirn. input events	Yes
	M7	wPon/of: press winding motor on/off input events	Yes
	T1	pPup/dn: press pusher retracted/not−retracted output events	No
	T4	wPup/dn: press weight raised/lowered output events	No
Unloading	M8	pcUon/of: unloading pusher and conveyor motor on/off input events	Yes
	L8	pUup/dn: unloading pusher retracted/not−retracted output events	No

Figure 6.1: Legend of signal and event labels

Also, the guard for the event a will be weakened as follows:

$$a : [(Tf \vee Tfson) \wedge (l \wedge k \wedge j \wedge i \wedge h \wedge g)] \Rightarrow [l \rightsquigarrow a].$$

The rules for the other events b, c, d, e and f will also be altered in a similar way.

- Initial conditions: Tf, Tr, $Tfson$, a, b, c, d, e, f = [`off, off, off, 0, 0, 0, 0, 0, 0`].
- Event occurrence rules:

$$\begin{aligned}
&a : [Tf \wedge (l \wedge k \wedge j \wedge i \wedge h \wedge g)] \Rightarrow [l \rightsquigarrow a];\\
&l : [Tr \wedge (a \wedge k \wedge j \wedge i \wedge h \wedge g)] \Rightarrow [a \rightsquigarrow l];\\
&b : [Tf \wedge (a \wedge k \wedge j \wedge i \wedge h \wedge g)] \Rightarrow [k \rightsquigarrow b];\\
&k : [Tr \wedge (a \wedge b \wedge j \wedge i \wedge h \wedge g)] \Rightarrow [b \rightsquigarrow k]\\
&c : [Tf \wedge (a \wedge b \wedge j \wedge i \wedge h \wedge g)] \Rightarrow [j \rightsquigarrow c];\\
&j : [Tr \wedge (a \wedge b \wedge c \wedge i \wedge h \wedge g)] \Rightarrow [c \rightsquigarrow j];\\
&d : [Tf \wedge (a \wedge b \wedge c \wedge i \wedge h \wedge g)] \Rightarrow [i \rightsquigarrow d];\\
&i : [Tr \wedge (a \wedge b \wedge c \wedge d \wedge h \wedge g)] \Rightarrow [d \rightsquigarrow i];\\
&e : [Tf \wedge (a \wedge b \wedge c \wedge d \wedge h \wedge g)] \Rightarrow [h \rightsquigarrow e];\\
&h : [Tr \wedge (a \wedge b \wedge c \wedge d \wedge e \wedge g)] \Rightarrow [e \rightsquigarrow h];\\
&f : [Tf \wedge (a \wedge b \wedge c \wedge d \wedge e \wedge g)] \Rightarrow [g \rightsquigarrow f];\\
&g : [Tr \wedge (a \wedge b \wedge c \wedge d \wedge e \wedge f)] \Rightarrow [f \rightsquigarrow g];\\
&Tfon : [\overline{Tf} \wedge \overline{Tr}] \Rightarrow [\overline{Tf} \rightsquigarrow Tf];\\
&Tfoff : [Tf \wedge \overline{Tr}] \Rightarrow [Tf \rightsquigarrow \overline{Tf}];\\
&Tron : [\overline{Tf} \wedge \overline{Tr}] \Rightarrow [\overline{Tr} \rightsquigarrow Tr];\\
&Troff : [\overline{Tf} \wedge Tr] \Rightarrow [Tr \rightsquigarrow \overline{Tr}].
\end{aligned}$$

Figure 6.2: Rules-based model of the transporter section

2. **Chassis:** The chassis conveyor conveys parts to its docking area. The chassis dock acts as a buffer with a capacity of one part. Parts are pushed off the dock onto an empty waiting transporter by the chassis pusher. Sensors monitor the retracted position of the pusher and presence of part on the dock.

The initial conditions of the chassis section require that the chassis conveyor motor (cC) and pusher motor (pC) be off, the pusher should be retracted (puC), there should be a part loaded on the conveyor (ld), and no part on the chassis dock (dC). Also there should be no jamming in the chassis section (x). The operation of the chassis section begins when the conveyor is turned on ($cCon$), the part rolls off the conveyor and is delivered to the chassis dock ($dCup$). Next, the pusher motor is turned on ($pCon$), which causes the pusher to operate ($puCdn$ and it pushes the part off the dock ($dCoff$) onto the transporter. The pusher returns to its original retracted position ($puCup$). The rules-based model of the chassis is given in Figure 6.3.

3. **Roof:** The rules-based model of the roof is given in Figure 6.4. The roof conveyor (cR) conveys parts that are loaded on it onto the roof dock which also has a buffer size of one. The part is pushed off the dock onto a waiting transporter by the roof pusher (pR). Sensors monitor the retracted position of the pusher (puR) and presence of part on the dock (dR). The operation of the roof is similar to that of the chassis section, with the exception that the presence of a part on the roof dock is indicated by $dRdn$. This is due to the fact that the roof is black in color and when positioned under a light sensor causes the $dRdn$ event. The chassis on the other hand is yellow in color and causes the $dCup$ event to occur when on the chassis dock.

   The actuators and sensors in various sections are subject to *stuck open/close* and *stuck up/down* faults respectively. In addition the sections could also encounter various system faults, such as power failure, software malfunctions, etc.

4. **Press:** The rules-based model of the press is given in Figure 6.5. The pressing of the roof and the chassis is done by releasing a heavy LEGO® block onto a properly positioned transporter carrying the roof-chassis combination. The mechanism is controlled by a press pusher and winding motor. Initially the pusher is advanced (pP) so that the weighted block is suspended at a certain

- Initial conditions: pC, cC, dC, x, ld, puC=[`off, off, dn, dn, up, up`].
- Event occurrence rules:

$$
\begin{aligned}
&puCup : [(\overline{cC} \wedge pC) \wedge (\overline{puC} \wedge \overline{dC} \wedge \overline{x} \wedge ld \wedge f) \\
&\qquad \vee (\overline{cC} \wedge pC) \wedge (\overline{puC} \wedge dC \wedge \overline{x} \wedge \overline{ld} \wedge f) \\
&\qquad \vee (cC \wedge pC) \wedge (\overline{puC} \wedge \overline{dC} \wedge \overline{x} \wedge \overline{ld} \wedge f)] \Rightarrow [\overline{puC} \rightsquigarrow puC]; \\
&puCdn : [(\overline{cC} \wedge pC) \wedge (puC \wedge \overline{dC} \wedge \overline{x} \wedge ld \wedge f) \\
&\qquad \vee (cC \wedge pC) \wedge (puC \wedge dC \wedge \overline{x} \wedge ld \wedge f) \\
&\qquad \vee (\overline{cC} \wedge pC) \wedge (puC \wedge \overline{dC} \wedge \overline{x} \wedge \overline{ld} \wedge f) \\
&\qquad \vee (cC \wedge pC) \wedge (puC \wedge \overline{dC} \wedge \overline{x} \wedge \overline{ld} \wedge f)] \Rightarrow [puC \rightsquigarrow \overline{puC}]; \\
&dCup : [(cC \wedge \overline{pC}) \wedge (puC \wedge \overline{dC} \wedge \overline{x} \wedge ld \wedge f)] \Rightarrow [\overline{dC} \rightsquigarrow dC]; \\
&dCdn : [(\overline{cC} \wedge pC) \wedge (\overline{puC} \wedge dC \wedge \overline{x} \wedge ld \wedge f) \\
&\qquad \vee (cC \wedge pC) \wedge (\overline{puC} \wedge dC \wedge \overline{x} \wedge ld \wedge f)] \Rightarrow [dC \rightsquigarrow \overline{dC}]; \\
&ldup : [(\overline{cC} \wedge \overline{pC}) \wedge (puC \wedge \overline{dC} \wedge \overline{x} \wedge \overline{ld})] \Rightarrow [\overline{ld} \rightsquigarrow ld]; \\
&lddn : [(\overline{cC} \wedge pC) \wedge (\overline{puC} \wedge \overline{dC} \wedge \overline{x} \wedge ld \wedge f) \\
&\qquad \vee (cC \wedge pC) \wedge (\overline{puC} \wedge \overline{dC} \wedge \overline{x} \wedge ld \wedge f)] \Rightarrow [ld \rightsquigarrow \overline{ld}]; \\
&x : [(cC \wedge pC) \wedge (puC \wedge \overline{dC} \wedge \overline{x} \wedge ld) \\
&\qquad \vee (cC \wedge pC) \wedge (\overline{puC} \wedge \overline{dC} \wedge \overline{x} \wedge ld) \\
&\qquad \vee (cC \wedge \overline{pC}) \wedge (\overline{puC} \wedge \overline{dC} \wedge \overline{x} \wedge ld)] \Rightarrow [\overline{x} \rightsquigarrow x]; \\
&\overline{x} : [\texttt{false}] \Rightarrow [x \rightsquigarrow \overline{x}]; \\
&pCon : [\overline{pC}] \Rightarrow [\overline{pC} \rightsquigarrow pC]; \qquad pCoff : [pC] \Rightarrow [pC \rightsquigarrow \overline{pC}]; \\
&cCon : [\overline{cC}] \Rightarrow [\overline{cC} \rightsquigarrow cC]; \qquad cCoff : [cC] \Rightarrow [cC \rightsquigarrow \overline{cC}].
\end{aligned}
$$

Figure 6.3: Rules-based model of the chassis section

- Initial conditions: pR, cR, dR, ld, x, puR = [`off, off, up, up, dn, up`].
- Event occurrence rules:

$$
\begin{aligned}
&puRup : [(\overline{cR} \wedge pR) \wedge (\overline{puR} \wedge dR \wedge \overline{x} \wedge ld) \\
&\qquad \vee (\overline{cR} \wedge pR) \wedge (\overline{puR} \wedge dR \wedge \overline{x} \wedge \overline{ld}) \\
&\qquad \vee (cC \wedge pR) \wedge (\overline{puR} \wedge dR \wedge \overline{x} \wedge \overline{ld})] \Rightarrow [\overline{puR} \rightsquigarrow puR]; \\
&puRdn : [(\overline{cR} \wedge pR) \wedge (puR \wedge dR \wedge \overline{x} \wedge ld) \\
&\qquad \vee (cR \wedge pR) \wedge (puR \wedge \overline{dR} \wedge \overline{x} \wedge ld) \\
&\qquad \vee (\overline{cR} \wedge pR) \wedge (puR \wedge dR \wedge \overline{x} \wedge \overline{ld}) \\
&\qquad \vee (cR \wedge pR) \wedge (puR \wedge dR \wedge \overline{x} \wedge \overline{ld})] \Rightarrow [puR \rightsquigarrow \overline{puR}]; \\
&dRup : [(\overline{cR} \wedge pR) \wedge (\overline{puR} \wedge \overline{dR} \wedge \overline{x} \wedge ld) \\
&\qquad \vee (cR \wedge pR) \wedge (\overline{puR} \wedge \overline{dR} \wedge \overline{x} \wedge ld \wedge f)] \Rightarrow [\overline{dR} \rightsquigarrow dR]; \\
&dRdn : [(cR \wedge \overline{pR}) \wedge (puR \wedge dR \wedge \overline{x} \wedge ld)] \Rightarrow [dR \rightsquigarrow \overline{dR}]; \\
&ldup : [(\overline{cR} \wedge \overline{pR}) \wedge (puR \wedge dR \wedge \overline{x} \wedge \overline{ld})] \Rightarrow [\overline{ld} \rightsquigarrow ld]; \\
&lddn : [(\overline{cR} \wedge pR) \wedge (\overline{puR} \wedge dR \wedge \overline{x} \wedge ld) \\
&\qquad \vee (cR \wedge pR) \wedge (\overline{puR} \wedge dR \wedge \overline{x} \wedge ld)] \Rightarrow [ld \rightsquigarrow \overline{ld}]; \\
&x : [(cR \wedge pR) \wedge (puR \wedge dR \wedge \overline{x} \wedge ld) \\
&\qquad \vee (cR \wedge pR) \wedge (\overline{puR} \wedge dR \wedge \overline{x} \wedge ld) \\
&\qquad \vee (cR \wedge \overline{pR}) \wedge (\overline{puR} \wedge dR \wedge \overline{x} \wedge ld)] \Rightarrow [\overline{x} \rightsquigarrow x]; \\
&\overline{x} : [\texttt{false}] \Rightarrow [x \rightsquigarrow \overline{x}]; \\
&pRon : [\overline{pR}] \Rightarrow [\overline{pR} \rightsquigarrow pR]; \qquad pRoff : [pR] \Rightarrow [pR \rightsquigarrow \overline{pR}]; \\
&cRon : [\overline{cR}] \Rightarrow [\overline{cR} \rightsquigarrow cR]; \qquad cRoff : [cR] \Rightarrow [cR \rightsquigarrow \overline{cR}].
\end{aligned}
$$

Figure 6.4: Rules-based model of the roof section

- Initial conditions: pPr, pPf, wP, wtP, pP, x = [`off, off, off, up, dn, dn`].
- Event occurrence rules:

$$
\begin{aligned}
&wtPup : [(pPf \wedge \overline{pPr} \wedge wP) \wedge (\overline{pP} \wedge \overline{wtP} \wedge \overline{x}) \\
&\qquad \vee (\overline{pPf} \wedge \overline{pPr} \wedge wP) \wedge (\overline{pP} \wedge \overline{wtP} \wedge \overline{x})] \Rightarrow [\overline{wtP} \rightsquigarrow wtP]; \\
&wtPdn : [(\overline{pPf} \wedge pPr \wedge \overline{wP}) \wedge (pP \wedge wtP \wedge \overline{x}) \\
&\qquad \vee (\overline{pPf} \wedge pPr \wedge wP) \wedge (pP \wedge wtP \wedge \overline{x}) \\
&\qquad \vee (\overline{pPf} \wedge \overline{pPr} \wedge wP) \wedge (pP \wedge wtP \wedge \overline{x}) \\
&\qquad \vee (pPf \wedge \overline{pPr} \wedge wP) \wedge (pP \wedge wtP \wedge \overline{x})] \Rightarrow [wtP \rightsquigarrow \overline{wtP}]; \\
&pPup : [(\overline{pPf} \wedge pPr \wedge wP) \wedge (\overline{pP} \wedge wtP \wedge \overline{x}) \\
&\qquad \vee (\overline{pPf} \wedge pPr \wedge \overline{wP}) \wedge (\overline{pP} \wedge wtP \wedge \overline{x}) \\
&\qquad \vee (\overline{pPf} \wedge pPr \wedge \overline{wP}) \wedge (\overline{pP} \wedge \overline{wtP} \wedge \overline{x}) \\
&\qquad \vee (\overline{pPf} \wedge pPr \wedge wP) \wedge (\overline{pP} \wedge \overline{wtP} \wedge \overline{x})] \Rightarrow [\overline{pP} \rightsquigarrow pP]; \\
&pPdn : [(pPf \wedge \overline{pPr} \wedge \overline{wP}) \wedge (pP \wedge \overline{wtP} \wedge \overline{x})] \Rightarrow [pP \rightsquigarrow \overline{pP}]; \\
&x : [(pPf \wedge \overline{pPr} \wedge wP) \wedge (pP \wedge \overline{wtP} \wedge \overline{x}) \\
&\qquad \vee (pPf \wedge \overline{pPr} \wedge wP) \wedge (pP \wedge wtP \wedge \overline{x})] \Rightarrow [\overline{x} \rightsquigarrow x]; \\
&\overline{x} : [\texttt{false}] \Rightarrow [x \rightsquigarrow \overline{x}]; \\
&pPfon : [\overline{pPr} \wedge \overline{pPf}] \Rightarrow [\overline{pPf} \rightsquigarrow pPf]; \\
&pPfoff : [pPf \wedge \overline{pPr}] \Rightarrow [pPf \rightsquigarrow \overline{pPf}]; \\
&pPron : [\overline{pPr} \wedge \overline{pPf}] \Rightarrow [\overline{pPr} \rightsquigarrow pPr]; \\
&wProff : [pPr \wedge \overline{pPf}] \Rightarrow [pPr \rightsquigarrow \overline{pPr}]; \\
&wPon : [\overline{wP}] \Rightarrow [\overline{wP} \rightsquigarrow wP]; wPoff : [wP] \Rightarrow [wP \rightsquigarrow \overline{wP}].
\end{aligned}
$$

Figure 6.5: Rules-based model of the press section

height (wtP). When the pusher motor is reversed (pPr) retracts the weight descends ($wtPdn$) and presses the pieces together. After this the pusher is advanced again so as to mesh with the winding motor (wP) gears, which when switched on raises the block up again. The retracted position of the pusher ($pPup$) and the raised position of the block ($wtPup$) are monitored by sensors.

The press section is the most complex section of the system and requires precise alignment of the pushing, lifting, and positioning mechanisms.

5. **Unloading:** The unloading conveyor (pcU) conveys parts that are pushed onto it by the unloading pusher (also pcU). There is a sensor for monitoring the retracted position of the pusher (pU). When a part has to be removed from the transporter positioned at the unloading section, the conveyor and pusher are turned on simultaneously ($pcUon$), and cause the pusher to advance ($pUdn$). After pushing the assembled roof-chassis the pusher returns to its retracted position ($pUup$) and the unloading pusher-conveyor is turned off ($pcUof$). The rules-based model of the transporter is given in Figure 6.6.

   As with any other motor in the system, the unloading conveyor-pusher motor can get stuck in the *on* or *off* positions. Also the sensors can be stuck in either their *up* or *dn* position. The rules-based model of the system can be altered accordingly to reflect these faults. For example, in the presence of $pcUsoF$ fault, the altered rule for the unloading pusher advancing ($pUdn$) is given as:

$$pUdn : [(pcU \vee pcUsoF) \wedge (pU \wedge l)] \Rightarrow [pU \rightsquigarrow \overline{pU}].$$

   Similarly, in the unloading section, a *motor power* fault signal, could cause the motor to stop. The events of the power fault are $powerF$ and $powerR$, denoting the power fault, and recovery from power fault events. Owing to the modeling of the *power* fault signal, the rules for the unloading motor event($pcUon$) and for the retracted/advanced positions of the pusher ($pUdn$, $pUup$ respectively are altered. For example, the altered rule for $pUdn$, having a $pcUsoF$ fault, and susceptible to a motor power fault, is given by:

$$pUdn : [(pcU \vee pcUsoF) \wedge \overline{power} \wedge (pU \wedge l)] \Rightarrow [pU \rightsquigarrow \overline{pU}].$$

We input the rules of each sub-system in $NuSMV$ software tool, followed by a synchronous composition of the models and model checking. The diagnosability test does not hold for this system, which means that this system are not diagnosable.

6.3 Diagnosis Technique Illustration for Rules-based Model

The following observation can be made about the detection of a fault: A fault is detected when a non-faulty guard condition is false, but the consequent event occurs. For this, all

- Initial conditions: pcU, $pcUsoF$, pU = [`off, off, dn`].
- Event occurrence rules:

$$pUdn : [(pcU \vee pcUsoF) \wedge (pU \wedge l)] \Rightarrow [pU \rightsquigarrow \overline{pU}];$$
$$pUup : [(pcU \vee pcUsoF) \wedge (\overline{pU} \wedge l)] \Rightarrow [\overline{pU} \rightsquigarrow pU];$$
$$pcUon : [\overline{pcU}] \Rightarrow [\overline{pcU} \rightsquigarrow pcU];$$
$$pcUof : [pcU \vee \overline{pcUsoF}] \Rightarrow [pcU \rightsquigarrow \overline{pcU}].$$

Figure 6.6: Rules-based model of the unloading section

traces indistinguishable to a sufficiently long extension of a trace containing the fault should themselves be faulty. This can be verified through a diagnosability test. For a diagnosable system, a diagnoser can be constructed to monitor observation sequence and report the occurrence of a fault.

For the purpose of illustrating the diagnosis technique, we use a simplified model of the transporter section of the LEGO® assembly-line (see Figure 6.7). This transporter moves between home and extended positions, crossing a number of intermediary positions. The events that can occur in transport system are: *Tfon, Tfoff, Tron, Troff, iup, idn, eup, edn. Tfon/Tfoff* refers to the gear-box motor being turned on/off in the forward direction, while *Tron/Troff* is for the reverse direction. When the transporter leaves the home position it enters an intermediary position, *i*, which is a collection of all those positions whose values are unimportant from the positioning point of view. The events *iup/idn* corresponds to the transporter arriving/leaving the intermediary position from/to the home position; and *eup/edn* corresponds to its arriving/leaving the extended position from/to the intermediate one.

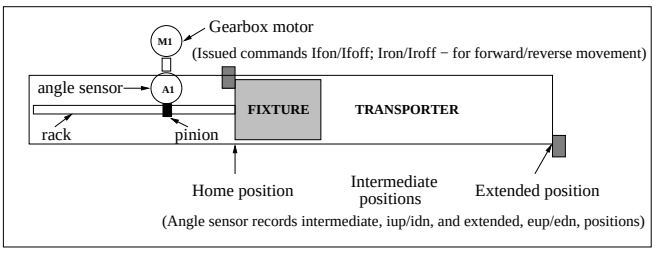

Figure 6.7: System layout

We assume that the initial state for the present system is when all the actuators are off (*Troff, Tfoff*) and the transporter is in home position (*idn, edn*). The model of the transporter in the rules-based formalism is given in Figure 6.8.

- Initial conditions: Tf, Tr, i, e = [`off, off, down, down`].
- Event occurrence rules:

$$iup : [Tf \wedge \overline{Tr} \wedge (\overline{i} \wedge \overline{e})] \Rightarrow [\overline{i} \rightsquigarrow i];$$
$$idn : [Tr \wedge \overline{Tf} \wedge (i \wedge \overline{e})] \Rightarrow [i \rightsquigarrow \overline{i}];$$
$$eup : [Tf \wedge \overline{Tr} \wedge (i \wedge \overline{e})] \Rightarrow [\overline{e} \rightsquigarrow e];$$
$$edn : [Tr \wedge \overline{Tf} \wedge (i \wedge e)] \Rightarrow [e \rightsquigarrow \overline{e}];$$
$$Tfon : [\overline{Tf}] \Rightarrow [\overline{Tf} \rightsquigarrow Tf]; \qquad Tfoff : [Tf] \Rightarrow [Tf \rightsquigarrow \overline{Tf}];$$
$$Tron : [\overline{Tr}] \Rightarrow [\overline{Tr} \rightsquigarrow Tr]; \qquad Troff : [Tr] \Rightarrow [Tr \rightsquigarrow \overline{Tr}].$$

Figure 6.8: Rules-based model of the transporter without faults

Next, we extend the transporter model to include a fault. Suppose the transport is prone to the forward motor Tf stuck open fault ($TfsonF$), whose occurrence does not alter the transport speed. The rules-based model of the transporter with this fault is given in Figure 6.9.

6.3.1 Diagnosability Test

For the transporter of Figure 6.7, we let $\vec{v} = (Tf, Tr, Tfson, i, e)$ denote the state variables of the transporter, and augment it with the boolean valued variable F to obtain the augmented state variable $\vec{x} = (\vec{v}, F)$. The augmented rules-based model of the transporter with $TfsonF$ fault is given in Figure 6.10.

Next using the state variable $\vec{x'} = (\vec{v'}, F')$ for the second copy of the augmented model, where $\vec{v'} = (Tf', Tr', Tfson', i', e')$, we compute the masked composition of the two augmented models. The resulting rules-based model is shown in Figure 6.11, where we have used σ to denote any of the following variables: *iup*, *idn*, *eup*, *edn*, $Tfon$, $Tfoff$, $Tron$ and $Troff$.

- Initial conditions: Tf, Tr, $Tfson$, i, e = [`off, off, off, down, down`].
- Event occurrence rules:

$$
\begin{aligned}
&iup : [(Tf \vee Tfson) \wedge \overline{Tr} \wedge (\overline{i} \wedge \overline{e})] \Rightarrow [\overline{i} \rightsquigarrow i];\\
&idn : [Tr \wedge \overline{Tf} \wedge (i \wedge \overline{e})] \Rightarrow [i \rightsquigarrow \overline{i}];\\
&eup : [(Tf \vee Tfson) \wedge \overline{Tr} \wedge (i \wedge \overline{e})] \Rightarrow [\overline{e} \rightsquigarrow e];\\
&edn : [Tr \wedge \overline{Tf} \wedge (i \wedge e)] \Rightarrow [e \rightsquigarrow \overline{e}];\\
&Tfon : [\overline{Tf}] \Rightarrow [\overline{Tf} \rightsquigarrow Tf];\\
&Tfoff : [Tf \vee Tfson] \Rightarrow [Tf \rightsquigarrow \overline{Tf}];\\
&Tron : [\overline{Tr}] \Rightarrow [\overline{Tr} \rightsquigarrow Tr];\\
&Troff : [Tr] \Rightarrow [Tf \rightsquigarrow \overline{Tr}];\\
&TfsonF : [Tf \wedge \overline{Tfson}] \Rightarrow [\overline{Tfson} \rightsquigarrow Tfson];\\
&TfsonR : [Tfson] \Rightarrow [Tfson \rightsquigarrow \overline{Tfson}].
\end{aligned}
$$

Figure 6.9: Rules-based model of the transporter with a $TfsonF$ fault

6.3.2 Diagnoser Synthesis

When the system is diagnosable, we can synthesize its on-line diagnoser. Otherwise, the system can be made diagnosable by sensor refinement using the technique developed in [50]. For example the transporter of Figure 6.7 prone to the forward motor stuck on fault, $TfsonF$, is not diagnosable. If we install a smart sensor that can sense the motor speed, then the occurrence of the $TfsonF$ can be declared when $Tfoff$ holds, but the motor speed is non-zero.

For a diagnosable system, we can compute a diagnoser using the method given in [44]. For this, we first obtain a rules-based model of the fault-free system by omitting the rules for the fault events (such as $TfsonF$ and $TfsonR$ for the transporter system). For each $k \geq 0$, the diagnoser maintains a predicate $N_k(\vec{v}) \in \mathcal{P}(\vec{v})$ that estimates the set of possible non-faulty states of the system following the occurrence of the kth observable event. $N_k(\vec{v})$ is computed iteratively as follows:

$$N_0(\vec{v}) = fr^*_{M^{-1}(\epsilon)\cap\Sigma} I(\vec{v}); \quad N_{k+1}(\vec{v}) = fr_{M^{-1}(\delta_k)}[fr^*_{M^{-1}(\epsilon)\cap\Sigma}(N_k(\vec{v}))],$$

where $\delta_k \in M(\Sigma) - \{\epsilon\}$ denotes the kth observation. A fault is said to have been detected

- Refined initial conditions: $I(\vec{x}) = I(Tf, Tr, Tfson, i, e, F) =$ [`off, off, off, down, down, 0`].
- Event occurrence rules:

$$
\begin{aligned}
&iup : [(Tf \vee Tfson) \wedge \overline{Tr} \wedge (\overline{i} \wedge \overline{e})] \Rightarrow [\overline{i}, F \rightsquigarrow i, F];\\
&idn : [Tr \wedge \overline{Tf} \wedge (i \wedge \overline{e})] \Rightarrow [i, F \rightsquigarrow \overline{i}, F];\\
&eup : [(Tf \vee Tfson) \wedge \overline{Tr} \wedge (i \wedge \overline{e})] \Rightarrow [\overline{e}, F \rightsquigarrow e, F];\\
&edn : [Tr \wedge \overline{Tf} \wedge (i \wedge e)] \Rightarrow [e, F \rightsquigarrow \overline{e}, F];\\
&Tfon : [\overline{Tf}] \Rightarrow [\overline{Tf}, F \rightsquigarrow Tf, F];\\
&Tfoff : [Tf \vee Tfson] \Rightarrow [Tf, F \rightsquigarrow \overline{Tf}, F];\\
&Tron : [\overline{Tr}] \Rightarrow [\overline{Tr}, F \rightsquigarrow Tr, F];\\
&Troff : [Tr] \Rightarrow [Tr, F \rightsquigarrow \overline{Tr}, F];\\
&TfsonF : [Tf \wedge \overline{Tfson}] \Rightarrow [\overline{Tfson}, F \rightsquigarrow Tfson, 1];\\
&TfsonR : [Tfson] \Rightarrow [Tfson, F \rightsquigarrow \overline{Tfson}, 0].
\end{aligned}
$$

Figure 6.10: Augmented Rules-based model of the transporter with a $TfsonF$ fault

- Initial conditions: $I(\vec{x},(\vec{x'}) = (I(\vec{v}),0,I(\vec{v'}),0)$.
- Event occurrence rules:

$(iup,iup) : [(Tf \vee Tfson) \wedge \overline{Tr} \wedge (\overline{i} \wedge \overline{e})] \Rightarrow [\overline{i},F,\overline{i'},F' \rightsquigarrow i,F,i',F'];$
$(idn,idn) : [Tr \wedge \overline{Tf} \wedge (i \wedge \overline{e})] \Rightarrow [i,F,i',F' \rightsquigarrow \overline{i},F,\overline{i'},F'];$
$(eup,eup) : [(Tf \vee Tfson) \wedge \overline{Tr} \wedge (i \wedge \overline{e})] \Rightarrow [\overline{e},F,\overline{e'},F' \rightsquigarrow e,F,e',F'];$
$(edn,edn) : [Tr \wedge \overline{Tf}) \wedge (i \wedge e)] \Rightarrow [e,F,e',F',\rightsquigarrow \overline{e},F\overline{e'},F'];$
$(Tfon,Tfon) : [\overline{Tf}] \Rightarrow [\overline{Tf},F,\overline{Tf'},F' \rightsquigarrow Tf,F,Tf',F'];$
$(Tfoff,Tfoff) : [Tf \vee Tfson] \Rightarrow [Tf,F,Tf',F' \rightsquigarrow \overline{Tf},F,\overline{Tf'},F'];$
$(Tron,Tron) : [\overline{Tr}] \Rightarrow [\overline{Tr},F,\overline{Tr'},F' \rightsquigarrow Tr,F,Tr',F'];$
$(Troff,Troff) : [Tr] \Rightarrow [Tr,F,Tr',F' \rightsquigarrow \overline{Tr},F,\overline{Tr'},F'];$
$(\sigma,TfsonF) : [Tf \wedge \overline{Tfson}] \Rightarrow [\overline{\sigma},F,\overline{\sigma'},F' \rightsquigarrow \sigma,F,\sigma',1];$
$(TfsonF,\sigma') : [Tf \wedge \overline{Tfson}] \Rightarrow [\overline{\sigma},F,\overline{\sigma'},F' \rightsquigarrow \sigma,1,\sigma',F'].$

Figure 6.11: Masked synchronization of two augmented rules-based model of the transporter

when $N_k(\vec{v}) = False$.

By computing different possible diagnoser states following all different possible observation sequences we can obtain the entire diagnoser. The result of such a computation for the transporter system is shown as an automaton in Figure 6.13. Any observation sequence that is not accepted by this diagnoser automaton indicates the occurrence of the $TfsonF$ fault.

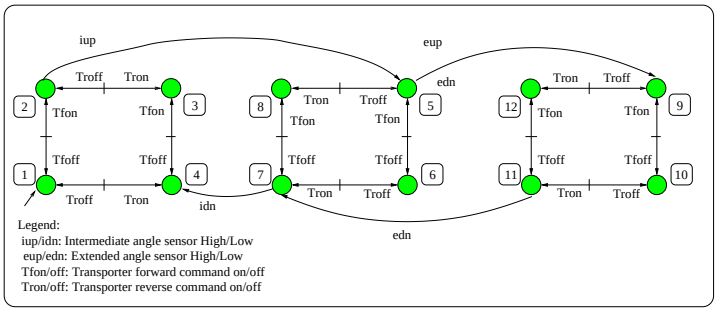

Figure 6.12: Diagnoser of the transporter with $TfsonF$ fault

6.3.3 Designing Diagnosable Systems

When the system is diagnosable, we can synthesize its on-line diagnoser. Otherwise, we can make it diagnosable by sensor refinement or sensor augmentation. For sensor refinement we can apply the technique of [50]. Also, smart sensors can be installed in the system which can make the observation of a fault possible. Consider, for example, the transporter of Figure 6.7 prone to a stuck-on fault in the transporter motor, denoted by $Tfson F$. We know from earlier analysis that the system is not diagnosable, when $Tfson F$ is an unobservable event. Now, if we install a sensor that can sense the motor speed, then the occurrence of the $Tfson F$ can be declared when $Tfoff$ holds, but the motor speed is non-zero.

In such a situation, we can compute a diagnoser for the transporter system using the method given in [44]. The result of such a computation is shown as an automaton in Figure 6.13. Any observation sequence that is not accepted by this diagnoser automaton indicates the occurrence of the $Tfson F$ fault.

Using NuSMV software tool for model-checking the condition for the diagnosability, we found that the transporter with $Tfson F$ fault is not diagnosable. The above example demonstrates that the symbolic technique for diagnosis developed in [44] allows for the diagnosability verification of practical systems.

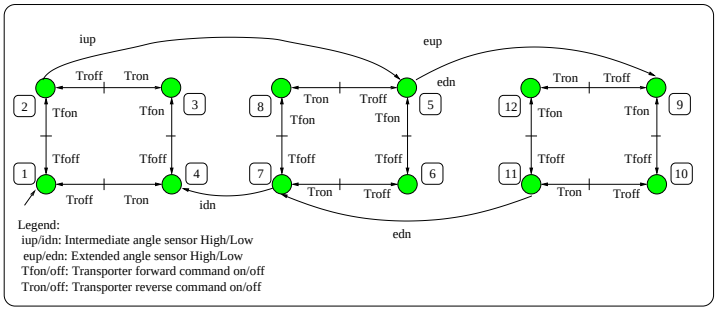

Figure 6.13: Diagnoser of the transporter with $Tfson F$ fault

In the absence of fault sensing devices, additional sensors may also be added to make the system diagnosable. Consider, for example, the transporter system, susceptible to a *TfsonF* fault. If the transporter is at the initial position $(\overline{i}, \overline{e})$, and is switched on, and then switched off prior to the occurrence of the *iup* event, then the *iup* event should not occur, unless there is a *TfsonF* in the system. Further, we can also examine the status of another sensor, e, which under precisely the same set of controllable events or faults, as those which cause the event *iup* to occur, will also cause the *eup* event to occur. However, neither *iup*, nor *eup* event

should occur, since the motor *Tf* has already been switched off when it was at the initial state). Thus, we monitor the extended position sensor, to ensure that it does not switch on, under these conditions as well. Having the additional e sensor will ensure that while it may be possible to have the *iup* event occur in an untimed model as part of its normal behavior, getting the *eup* event would indicate a $TfsonF$ fault to the diagnoser. If it does, then we can declare a $TfsonF$. Retaining, or adding the e sensor if one were not already present, thus makes the $TfsonF$ diagnosable, even without adding a smart fault sensor. Hence, the presence of the *Tfson* fault can be deduced, even though

$$\exists \vec{x_0}, \vec{y_0}[EGF(\vec{x} = \vec{x_0} \bigwedge \vec{y} = \vec{y_0} \bigwedge [(B(\vec{x_0}) \bigwedge \neg B(\vec{y_0}))])];$$

remains true and the system is not diagnosable.

Further, in some situations involving faults, in which we are looking for a signal to occur which never can (since the monitored sensor may be faulty), additional sensors should be inserted, which would be triggered by precisely the same controllable and uncontrollable faults events conditions. The presence of such, possibly redundant, sensors can be used for identifying certain faults.

6.4 Conclusion

The rules-based modeling formalism of [12] has been used to model and study diagnosis of a simple assembly-line built using LEGO® blocks. The advantage of using rules-based model is its compactness since it uses variables to represent states. The number of rules in the rules-based model is polynomial in number of system signals and faults. Symbolic technique for failure analysis, based on 1^{st}-order temporal logic model-checking, has been employed successfully. Existing software tools such as NuSMV software tool aid the analysis for systems with bounded state-space. If the given system is not diagnosable, refining the observation mask [50] makes the system diagnosable, and subsequently an on-line diagnoser for the system can be constructed. This was demonstrated using several examples drawn from the LEGO® assembly-line.

Chapter 7

Conclusions and Future Work

7.1 Conclusion

In this dissertation, Finite state machines (FSMs) are used for modeling operations of the assembly line built using LEGO® blocks , and for the specifications that accomplish the task of successfully completing the assembly repeatedly. Using the technique of Supervisory control theory (SCT), we derive a supervisor that enforces the specifications while offering the maximum flexibility of assembly. Subsequently a controller is extracted from the maximally permissive supervisor for the purpose of implementing the control by selecting, when possible, only one controllable event from among the ones allowed by the supervisor. Testing to check the correctness of the control code is reduced, since the controller is guaranteed to enforce the specifications.

Rules-based model has been employed in order to aid the rapid development of an accurate system model with faults which can subsequently be used for fault diagnosis. It relies on establishing rules for all the events in the system. This intuitive way of modeling systems using a set of rules can be implemented very easily. Any changes in the structure of the system, such as when actuators or sensors in the system are added or removed can easily be incorporated in the rules as well. Any added or removed faults can be incorporated in the rules easily. Once the rules for the DESs have been developed it can be used for fault diagnosis and diagnoser construction. The rules-based modeling formalism can be applied to a wide variety of discrete event systems, both timed and untimed, with or without faults, and having multi-valued signals, and ones possessing DESs abstractions. The main feature of the rules based modeling formalism is its size, which is polynomial in the number of system

signals.

In this dissertation we also study diagnosis of DESs. We gave a method for testing the diagnosability in automaton setting. We developed symbolic techniques for testing diagnosability and computing a diagnoser in rules based modeling formalism. Diagnosability test is shown to be an instance of 1^{st} order temporal logic model-checking. An on-line algorithm for diagnoser synthesis is obtained by using predicates and predicate transformers. We modeled an automated car assembly-line in rules-based modeling formalism with faults and used symbolic technique for failure analysis based on 1^{st} order temporal logic model-checking.

7.2 Future work

Some of the possible extensions to the modeling formalism are:

- Implementation of the supervisory control theory has been demonstrated by way of the control of a miniature assembly line built from LEGO® blocks. The main issue of complexity may be dealt with using modularity as we have demonstrated. A controller may be extracted from a maximally permissive supervisor either ad-hocly, or if need be, more systematically using optimal control.

- The derivation of plant models has been at the physical level, with signals and events. When working at a higher level of abstraction, the top level events are the macro events with regards to the lower, i.e., physical, level ones. A scheme for such a hierarchical rules based modeling could be developed for further reducing the complexity involved in DESs modeling.

- In the rules based modeling formalism the plant is modeled using rules, whereas in supervisory control algorithms both the plant and the control specification are modeled as automata. This forces the rules based model to be converted into an equivalent automata model before any supervisory control algorithms can be used. Modeling the control specifications as rules is in many cases not practical. Efficient ways of representing the specification such that the compact form of the rules based model can be used for directly computing the supervisor for the plant, is an area of for future investigation.

- A system with failure is not necessarily stable with respect to non-failure states since once the system enters the failure region it may stay there forever. However, a system

may be stabilizable in the sense when recovery to legal states may be possible. So, we need to diagnose the failure when it occurs and enable the corresponding recovery event so as to return to the normal region. This is called failure diagnosis and recovery and is an area for the future investigation.

- Extension of 1^{st}-order model-checking method to diagnosis of repeated, intermittent failures, real-time, and probabilistic systems is a possible future research direction.
- Further applications and software tool development.

Appendix A

NuSMV Programming For the Diagnosability Check of Maze Example

The NuSMV system is a tool for checking finite state systems against specifications in the temporal logic CTL and LTL. The input language of NuSMV is designed to allow the description of finite state systems that range from completely synchronous to asynchronous, and from the detailed to the abstract. The language provides for modular hierarchical descriptions, and for the definition of reusable components.

The following is the NuSMV programme for the diagnosability check of maze example.

```
MODULE main
VAR
mainInState:nodeZero;
stateOut: node;

rule1:ruleOrignal(mainInState);
ruleA:rule(rule1.stateOut,rule1.o1,rule1.o2,rule1.o3);
ruleB:rule(rule1.stateOut,rule1.o1,rule1.o2,rule1.o3);

LTLSPEC  F(G((ruleA.states.label)=(ruleB.states.label)))

MODULE node
```

```
VAR
state:{0,1,2,3};
label:boolean;

MODULE nodeZero
VAR
state:{0,1,2,3};
label:boolean;

ASSIGN
init(state):=0;
init(label):=0;

MODULE ruleOrignal(stateIn)
VAR
  states: node;
  stateOut:node;
  o1:boolean;
  o2:boolean;
  o3:boolean;

ASSIGN
  init(states.state):=stateIn.state;
  init(states.label):=stateIn.label;

  next(states.state):=
 case
   (states.state=0):case
       ((o1=1)&(o2=0)&(o3=0)):3;
       ((o1=0)&(o2=0)&(o3=1)):3;
       esac;

   (states.state=3):case
```

```
        ((o1=0)&(o2=1)&(o3=0)):0;
        ((o1=0)&(o2=0)&(o3=1)):3;
        esac;

    (states.state=2):case
        ((o1=0)&(o2=0)&(o3=1)):3;
        esac;

    (states.state=1):case
         1:2;
        esac;
 esac;

  next(states.label):=
 case
    ((states.state=0)&(states.label=0)):case
        ((o1=1)&(o2=0)&(o3=0)):0;
        ((o1=0)&(o2=0)&(o3=1)):1;
        esac;

    ((states.state=3)&(states.label=0)):case
        ((o1=0)&(o2=1)&(o3=0)):0;
        ((o1=0)&(o2=0)&(o3=1)):0;
        esac;

    (states.state=1):1;
    (states.label=1): 1;
 esac;

  stateOut.state:=states.state;
  stateOut.label:=states.label;

MODULE rule(stateIn,o1,o2,o3)
```

```
VAR
  states: node;
  stateOut:node;

ASSIGN
  init(states.state):=stateIn.state;
  init(states.label):=stateIn.label;

  next(states.state):=
 case
   (states.state=0):case
       ((o1=1)&(o2=0)&(o3=0)):3;
       ((o1=0)&(o2=0)&(o3=1)):3;
       esac;

   (states.state=3):case
       ((o1=0)&(o2=1)&(o3=0)):0;
       ((o1=0)&(o2=0)&(o3=1)):3;
       esac;

   (states.state=2):case
       ((o1=0)&(o2=0)&(o3=1)):3;
       esac;

   (states.state=1):case
        1:2;
       esac;
 esac;

  next(states.label):=
 case
   ((states.state=0)&(states.label=0)):case
       ((o1=1)&(o2=0)&(o3=0)):0;
```

```
        ((o1=0)&(o2=0)&(o3=1)):1;
        esac;

    ((states.state=3)&(states.label=0)):case
        ((o1=0)&(o2=1)&(o3=0)):0;
        ((o1=0)&(o2=0)&(o3=1)):0;
        esac;

     (states.state=1):1;
    (states.label=1): 1;
 esac;

  stateOut.state:=states.state;
  stateOut.label:=states.label;
```

The result after executing the program is:

```
$nusmv catMourseRules.smv
*** This is NuSMV2.1.2 (compiled 2002-11-22 12:00:00)
*** For more information of NuSMV see http://nusmv.irst.itc.it
*** or email to nusmv-users@irst.itc.it.
*** Please report bugs to <nusmv-users@irst.itc.it>.

-- specification A F G ruleA.states.label=ruleB.states.label is true
which means that the maze system is diagnosable.
```

Bibliography

[1] K. E. Arzen. Grafcet for intelligent supervisory control. *Automatica*, 30(10):1513–25, 1994.

[2] R. L. Aveyard. A boolean model for a class of discrete event systems. *IEEE Transactions on Systems, Man, and Cybernetics*, 4:249–258, 1974.

[3] S. Balemi, G. J. Hoffmann, P. Gyugyi, H. Wong-Toi, and G. F. Franklin. Supervisory control of a rapid thermal multiprocessor. *IEEE Transactions on Automatic Control*, 38(7):1040–1059, July 1993.

[4] M. Barbeau, F. Kabaza, and R. St.-Denis. A method for the synthesis of controllers to handel safety, liveness, and real-time constraints. *IEEE Transactions on Automatic Control*, 43(11):1543–1559, 1998.

[5] S. Bavishi and E. Chong. Automated fault diagnosis using a discrete event systems framework. In *Proceedings of 1994 IEEE International Symposium on Intelligent Control*, pages 213–218, 1994.

[6] A. Bouloutas, G. W. Hart, and M. Schwartz. On the design of observers for fault detection in communication networks. In A. Kershenbaum and et al., editors, *Network Management and Control*, pages 319–338. Plenum Press, 1990.

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

[8] C. Bousson, P. Gaborit, and M. Ghallab. Situation recognition: Representation and algorithms. In *Proceedings of the 13th IJCAI*, 1993.

[9] B. A. Brandin. The real-time supervisory control of an experimental manufacturing cell. *IEEE Transactions on Robotics and Automation*, 12(1):1–14, February 1996.

[10] C. G. Cassandras, J. Bergendahl, D. Esterman, and M. Sullivan. Computer controlled lego factory. Technical report, University of Massachusetts, Boston, MA, 1995.

[11] V. Chandra, Z. Huang, and R. Kumar. Discrete event modeling and control of an assembly line built using LEGO blocks. *IEEE Transactions on Systems, Man, and Cybernetics: Part C*, 2003. Accepted.

[12] V. Chandra and R. Kumar. A event occurrence rules based compact modeling formalism for a class of discrete event systems. *Mathematical and Computer Modeling of Dynamical Systems*, 8(1):49–73, 2002.

[13] Y. L. Chen and G. Provan. Fault diagnosis in timed discrete-event systems. In *Proceedings of the 38th IEEE Conference on Decision and Control*, pages 1756–1761, Pheonix, AZ, 1999.

[14] Yi-Liang Chen and G.Provan. Modeling and diagnosis of timed discrete event systems - a factory automation example. In *Proceedings of the American Control Conference ACC97*, pages 31–36, Albuquerque, New Mexcio, June 1997.

[15] A. Cimatti, E. M. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. Nusmv 2: An opensource tool for symbolic model checking. In *Proceeding of International Conference on Computer-Aided Verification*, Copenhagen, Denmark, July 2002.

[16] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, Cambridge, MA, 1999.

[17] L. Console, L. Portinale, D. Dupre, and P. Torasso. Diagnostic reasoning across different time points. In *Proceedings of 10th European conference On Artificial Intelligence ECAI 92*, volume 3, 1992.

[18] L. Console, L. Portinale, D. Dupre, and P. Torasso. Diagnosing time - varying misbehavior: an approach based on model decomposition. *Annals-of-Mathematics-and-Artificial-Intelligence*, 11, 1994.

[19] C. F. Cooper. Nestor: a computer based medical diagnostic aid that integrates causal and probabilistic knowledge. Technical Report HHP-84-48, Stanford University, CA, 1984.

[20] A. Darwiche and G. Provan. Exploiting system structure in model-based diagnosis of discrete event systems. In *Proceedings of the Seventh International Workshop on Principles of Diagnosis*, Val Morin Canada, 1996.

[21] S. R. Das and L. E. Holloway. Learning of time templates from system observation. In *Proceedings of the American Control Conference (ACC)*, volume 4, pages 2626–2630, Seattle, Washington, 1995.

[22] S. R. Das and L. E. Holloway. Characterizing a confidence space for discrete event timings for fault monitoring using discrete sensing and actuation signals. *IEEE Transactions on Systems, Man, and Cybernetics—Part A: Systems and Humans*, 30(1):52–66, 2000.

[23] R. Debouk, S. Lafortune, and D. Teneketzis. Coordinated decentralized protocols for failure diagnosis of discrete event systems. *Discrete Event Dynamical Systems: Theory and Applications*, 10:33–79, 2000.

[24] J. S. Breese E. J. Horwitz and M. Henrion. Decision theory in expert systems and artificial intelligence. *International Journal of Approximate Reasoning*, 1988.

[25] M. S. Elliott. Computer-assisted fault tree construction using a knowledge-based approach. *IEEE transactions On Reliability*, 43(1), 1994.

[26] E. A. Emerson. Temporal and modal logic. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*. Elsevier Science Publishers, 1990.

[27] A. et. al. Asse. Diagnosis based on subjective information in a solar energy plant. In E. Sanchez and L. A. Zadeh, editors, *Approximate reasoning in intelligent systems, decision and control*. Pergmon Press, 1988.

[28] P. M. Frank. Advances in observer-based fault diagnosis. In *Survey Paper Tooldiagnosis '93 (International Conference on Fault Diagnosis)*, Toulouse, France, 1993.

[29] B. Freyermuth. Knowledge based incipienct fault diagnosis of industrial robots. In *Proceedings of SAFEPROCESS' 91 International Conference on Fault Detection, Supervision and Safety for Technical Processes*, volume 2, Baden-Baden, Germany, September 1991.

[30] Johann Gamper. *A Temporal Reasoning and Abstraction Framwork for Model-Based Diagnosis Systems*. PhD thesis, RWTH, Aachen, Germany, 1996.

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

[32] É. Grégoire and D. Ansart. Detection of the main failure in complex critical systems. In *DISCRETE EVENT SYSTEMS Analysis and Control*, pages 363–370. Kluwer Academic Publishers, Boston, MA, 2000.

[33] W. A. Gruver and J. C. Boudreaux, editors. *Intelligent Manufacturing: Programming environments for CIM*. Advanced manufacturing Series. Springer-Verlag, New York, 1993.

[34] Walter Hamscher, Luca Console, and Johan De Kleer, editors. *Readings in Model-Based Diagnosis*. Morgan Kaufmann Publishers, 1992.

[35] D. M. Himmelblau. Fault detection and diagnosis in chemical and petrochemical process. *Chemical Engnieering*, 8, 1978.

[36] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, Inc., Englewood Cliffs, NJ, 1985.

[37] L. E. Holloway. On-line fault monitoring of a class of hybrid systems using templates with dynamic time scaling. In *Hybrid Systems III*. Springer-Verlag, Berlin, Germany, New Brunswick, NJ, USA, 1995.

[38] L. E. Holloway and S. Chand. Distributed fault monitoring in manufacturing systems using concurrent discrete-event observations. *Integrated Computer-Aided Engineering*, 3(4):244–254, 1996.

[39] L.E. Holloway and S. Chand. Time templates for discrete event fault monitoring in manufacturing systems. In *Proceedings of the American Cotrol Control Conference (ACC)*, volume 1, Baltimore, Maryland, 1994.

[40] R. A. Howard and J. E. Matherson. Influence diagrams. In Howard and Matherson, editors, *Readings in Decision Analysis*. Menlo Park, CA, 1981.

[41] A. Hsu, F. Eskafi, S. Sachs, and P. Varaiya. Protocol design for an automated highway system. *Discrete Event Dynamical Systems: Theory and Application*, 2(3/4):183–206, 1993.

[42] G. E. Hugghes and M. J. Creswell. *Introduction to Modal Logic*. Methuen, London, 1977.

[43] Z. Hunag, V. Chandra, S. Jiang, and R. Kumar. Modeling discrete event systems with faults using a rules-based modeling formalism. In *Proceedings of 2002 Conference on Decision and Control*, pages 4012–4017, Las Vegas, NV, December 2002.

[44] Z. Hunag, V. Chandra, S. Jiang, and R. Kumar. Diagnosis of discrete event systems in rules-based model using first order logic. In *Proceedings of 2003 Conference on Decision and Control*, Maui, HA, December 2003.

[45] R. Isermann. Fault diagnosis of machines via parameter estimation and knowledge processing. *Automatic*, 29(4), 1993.

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

[47] S. Jiang and R. Kumar. Failure diagnosis of discrete event systems with linear-time temporal logic fault specifications. *IEEE Transactions on Automatic Control*, 2001. Submitted.

[48] S. Jiang and R. Kumar. Supervisory control of discrete event systems with CTL* temporal logic specification. In *2001 IEEE Conference on Decision and Control*, pages 4122–4127, FL, December 2001.

[49] S. Jiang and R. Kumar. Diagnosis of repeated failures for discrete event systems with linear-time temporal logic specifications. *IEEE Transactions on Systems, Man, and Cybernetics: Part B*, 2002. Submitted.

[50] S. Jiang, R. Kumar, and H. E. Garcia. Optimal sensor selection for discrete event systems under partial observation. *IEEE Transactions on Automatic Control*, 48(3):369–381, March 2003.

[51] J. Kitowski and M. Bargiel. Diagnostics of faulty states in complex physical systems using fuzzy relational equaltions. In E. Sanchez and L. A. Zadeh, editors, *Approximate reasoning in intelligent systems, decision and control*. Pergmon Press, 1988.

[52] J. F. Knight and K. M. Passino. Decidability for a temporal logic used in discrete-event system analysis. *International Journal of Control*, 52(6):1489–1506, 1990.

[53] B. Kosko. *Neural networks: a dynamic systems approach to machine learning*. Preentice-Hall, Engelwood cliffs, NJ, 1992.

[54] R. Kumar and V. K. Garg. *Modeling and Control of Logical Discrete Event Systems*. Kluwer Academic Publishers, Boston, MA, 1995.

[55] R. Kumar, V. K. Garg, and S. I. Marcus. On controllability and normality of discrete event dynamical systems. *Systems and Control Letters*, 17(3):157–168, 1991.

[56] R. Kumar, V. K. Garg, and S. I. Marcus. Predicates and predicate transformers for supervisory control of discrete event systems. *IEEE Transactions on Automatic Control*, 38(2):232–247, February 1993.

[57] R. Kumar, S. Nelvagal, and S. I. Marcus. A discrete event systems approach for protocol conversion. *Discrete Event Dynamical Systems: Theory and Applications*, 7(3):295–315, 1997.

[58] S. Lapp and G. Powers. Computer aided synthesis of fault trees. *IEEE transactions On Reliability*, 26(1), 1977.

[59] M. Larsson. *Behavioral and structural model based approaches to discrete diagnosis*. PhD thesis, Linkoping University, Linkoping, Sweden, 1999.

[60] G. E. Lasker. Systems diagnostics: basic concepts and methodology. In *Proceedings of the SGSR Detroit Conference on World Problems and Systems Learning*, pages 749–769, 1983.

[61] S. Lauritzen and D. Spiegelhalter. Local computations with probabilities on graphical structures and their application to expert systems. *J. R. Statistical Society*, 50:157–224, 1988.

[62] Ryan J. Leduc. PLC implementation of a DES supervisor for a manufacturing testbed : An implementation perspective. Master's thesis, Department of Computer and Electrical Engineering, University of Toronto, Toronto, Canada, 1996.

[63] F. Lin. Analysis and synthesis of discrete event systems using temporal logic. *Control Theory and Advanced Technologies*, 9(1):341–350, 1993.

[64] F. Lin. Diagnosability of discrete event systems and its applications. *Discrete Event Dynamic Systems: Theory and Applications*, 4(1):197–212, 1994.

[65] F. Lin, J. Markee, and B. Rado. Design and test of mixed signal circuits: a discrete event approach. In *Proceedings of the 32nd IEEE Conference on Decision and Control*, pages 246–251, 1993.

[66] F. Lin and W. M. Wonham. On observability of discrete-event systems. *Information Sciences*, 44(3):173–198, 1988.

[67] Feng Lin. Diagnosability of discrete-event systems and its applications. *Discrete Event Dynamic systems*, 4, 1994.

[68] J.-Y. Lin and D. Ionescu. Verifying a class of nondeterministic discrete event systems in a generalized temporal logic. *IEEE Transactions on Systems, Man and Cybernetics*, 22(6):1461–1469, 1992.

[69] J.-Y. Lin and D. Ionescu. Reachability synthesis procedure for discrete event systems in a temporal logic. *IEEE Transactions on Systems, Man and Cybernetics*, 24(9):1397–1406, 1994.

[70] L. Ljung, editor. *System Identification: Theory for the User*. Prentice-Hall, 1999.

[71] J. Lygeros, D. N. Godbole, and M. Broucke. A fault tolerant control architecture for automated highway system. *IEEE Transactions on Control Systems Technology*, 8(2):205–219, March 2000.

[72] G. Michel. *Programmable Logic controllers, Architecture and Applications*. Wiley, NY, 1990.

[73] R. Milne, C. Nicol, M. Ghallab, L. Trave-massuyes, C.Bousson, J.Quevedo, C.Dousson, J. Aguilar, and A. Guasch. tiger: real-time situation assessment of dynamic systems. *Intelligent Systems Engineering*, 1994.

[74] M. Nybeg. *Model Based Fault Diagnosis: Methods, Theory and Automotive Engine Applications*. PhD thesis, Linkoping University, Linkonping, sweden, 1999.

[75] J .S. Ostroff. Synthesis of controllers for real-time discrete event systems. In *Proceedings of 28th IEEE Conference on Decision and Control*, Tampa, FL, 1989.

[76] J. S. Ostroff and W. M. Wonham. A framework for real-time discrete event control. *IEEE Transactions on Automatic Control*, 35(4):386–397, 1990.

[77] D. Pandalai and L. Holloway. Template languages for fault monitoring of timed discrete event processes. *IEEE Transactions on Automatic Control*, 45(5):868–882, May 2000.

[78] D.N. Pandalai and L.E. Holloway. Condition templates: Improved distributed models for automated fault monitoring of manufacturing systems. In *Proceedings of the International conference on Robotics and Automation*, pages 515–520, Minneapolis, Minnesota, 1996.

[79] Y. Park and E. K. P. Chong. Distributed inversion in timed discrete event systems. *Discrete Event Dynamic Systems: Theory and Applications*, 5(2-3):219–241, 1995.

[80] K. M. Passino and P. J. Antsaklis. Branching time temporal logic for discrete event system analysis. In *Proceedings of 1988 Allerton Conference*, pages 1160–1169, Allerton, IL, 1988.

[81] R. J. Patton. Robust model-based fault diagnosis: The state of the art. In *Proceedings of the IFAC Symposium of Fault Detection, Supervision and Safety for Technical Processes SAFEPROCESS94, IFAC Fault Detection, Supervision and Safety for Technical Processes*, Espoo, Finland, 1994.

[82] R. J. Patton, editor. *Robust Model-Based Fault Diagnosis for Dynamic System*. Kluwer Academic Publisher, 1999.

[83] J. Pearl. Fusion, propagation and structuring in belief networks. *Artificial Intelligence*, 29:241–288, 1986.

[84] C. Pecheur and A. Cimatti. Formal verification of diagnosability via symbolic model checking. In *Workshop on Model Checking and Artificial Intelligence*, Lyon, France, 2002.

[85] C. Perrow, editor. *Normal Accidents Living with High Risk Technologies*. Basic Books Inc., New York, 1984.

[86] A. Pnueli. The temporal logic of programs. In *Proceedings of 18th Annual Symposium on Foundations of Computer Science*, pages 46–57, Providence, RI, November 1977.

[87] David Poole. Explanation and prediction: An architecture for default and abductive reasoning. *Computation Intelligence*, 5(2), 1989.

[88] P. J. Ramadge and W. M. Wonham. Supervisory control of a class of discrete event processes. *SIAM Journal of Control and Optimization*, 25(1):206–230, 1987.

[89] P. J. Ramadge and W. M. Wonham. The control of discrete event systems. *Proceedings of IEEE: Special Issue on Discrete Event Systems*, 77:81–98, 1989.

[90] J. Rasmussen. Diagnostic reasoning in action. *IEEE transactions On System, Man and Cybernetics*, 23(4):981–991, 1993.

[91] R. Reiter. A theory of diagnosis from first principles. *Artifical Intelligence*, 32(1), 1987.

[92] T. Ruokonen, editor. *IFAC Symposium of Fault Detection, Supervision and Safety for Technical Processes (SAFEPROCESS 94)*. Espoo, Helsinki, Finland, 1994.

[93] R. H. Kwong S. H. Zad and W. M. Wonham. Fault diagnosis in timed discrete-event systems. In *Proceedings of the 38th IEEE Conference on Decision and Control*, pages 1756–1761, Pheonix, AZ, 1999.

[94] F. V. Jensen S. K. Andersen, G. k. Olesen and F. Jensen. Hugin - a shell for building belief universes for expert systems. In *Proceedings 11th International Joint Conference on Artificial Intelligence*, Detroit, 1989.

[95] M. Sampath. *A Discrete Event Systems Approach to Failure Diagnosis*. PhD thesis, Departement of Electrical Engineering and Computer Science, University of Michigan, Ann Arbor, MI, 1995.

[96] M. Sampath and S. Lafortune. Active diagnosis of discrete event systems. *IEEE Transactions on Automatic Control*, 43(7):908–929, 1998.

[97] M. Sampath, R. Sengupta, S. Lafortune, K. Sinaamohideen, and D. Teneketzis. Failure diagnosis using discrete event models. Technical report, Department of Electrical Engineerig and Computer Science, The University of Michigan, Ann Arbor, USA, May 1994.

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

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

[100] E. Sanchez. Medical diagnostics applications in a linguistic approach using fuzzy logic. In *Proceedings of the Int. Workshop on Fuzzy System Applications*, pages 38–50, Iizuka, Japan, 1988.

[101] W. T. Scherer and C. C. White. a survey of expert systems for equipment maintenance and diagnostics. In *Knowledge-based system diagnosis, supervision and control*, pages 285–300. Plenum, New York, 1989.

[102] K. T. Seow and R. Devanathan. Temporal framework for assembly sequence representation and analysis. *IEEE Transactions on Robotics and Automation*, 10(2):220–229, April 1994.

[103] K. T. Seow and R. Devanathan. A temporal logic approach to discrete event control for the safety cannonical class. *Systems and Control Letters*, 28:205–217, 1996.

[104] J. G. Thistle, R. P. Malhame, H.-H. Hoang, and S. Lafortune. Supervisory control of distributed systems part I: modeling, specification, and synthesis. Technical Report EPM/RT-97/08, Ecole Polytechnique de Montreal, 1997.

[105] J. G. Thistle and W. M. Wonham. Control problems in temporal logic framework. *International Journal of Control*, 44(4):943–976, 1986.

[106] Y. Tsukamoto and T. Terano. Failure diagnosis by using fuzzy logic. In *Proceedings of IEEE Conference on Decision Making and Control*, volume 4, pages 1390–1395, New Orlean, 1977.

[107] S. Tzafestas and K. Watanabe. Modern approaches to system/sensor fault detection and diagnosis. *Journal A*, 31(4), 1990.

[108] P. Varaiya. Smart cars on smart roads: problems of control. *IEEE Transactions on Automatic Control*, 38(2):195–207, 1993.

[109] N. Viswanadham and T. L. Johnson. Fault detection and diagnosis of automated manufacturing systems. In *In proceedings of the 27th IEEE Conference on Decision and Control (CDC)*, volume 3, pages 2301–2306, 1988.

[110] R. D. Vries. An automated methodology for generating a fault tree. *IEEE transactions On Reliability*, 39(1), 1990.

[111] G. Westerman, R. Kumar, C. Stroud, and J. R. Heath. Discrete event systems approach for delay fault analysis in digital circuits. In *Proceedings of 1998 American Control Conference*, Philadelphia, PA, 1998.

[112] G. Westerman, R. Kumar, C. Stroud, and J. R. Heath. Discrete event systems approach for delay fault analysis in digital circuits. In *Proceedings of 1998 American Control Conference*, Philadelphia, PA, 1998.

[113] B. Williams and P. Nayak. A model-based approach to reactive self-configuring systems. In *Proceedings of the 13th National conference On Artificial Intelligence (AAAI-96) Building construction*, 1996.

[114] Alan S. Willsky. A survey of design methods for failure detection in dynamic system. *Automatica*, 12, 1976.

[115] H. Wong-Toi and D. L. Dill. Synthesizing processes and schedulers from temporal specifications. In *Proceedings of the 1991 Computer-Aided Verification Workshop, (Lecture Notes in Computer Science)*, volume 531. Springer-Verlag, 1991.

[116] W. M. Wonham and P. J. Ramadge. On the supremal controllable sublanguage of a given language. *SIAM Journal of Control and Optimization*, 25(3):637–659, 1987.

[117] S. Young and V. K. Garg. Model uncertainty in discrete event systems. *SIAM Journal of Control and Optimization*, 33(1):208–226, 1995.

[118] S. H. Zad. *Fault diagnosis in discrete-event and hybrid systems.* PhD thesis, University of Toronto, Toronto, Canada, 1999.

[119] H. J. Zimmermann. *Fuzzy set theory and its applications.* Kluwer Academic Publishers, Boston, 1991.

Vita

Zhongdong Huang was born October 26, 1971, in Guangchang, P. R. China. He received the B.S. degree in electrical engineering from the Southern Institute of Metallurgy, Ganzhou, China, the M.S. degree in electrical engineering from Northeastern University, Shenyang, China, in 1992, 1995, respectively. He then worked with Electric Power Research Institute (EPRI), Beijing, China. He joined the graduate program in Electrical Engineering at the University of Kentucky, Lexington, USA in the summer, 1999, where he received a M.S. degree in Electrical and Computer Engineering in 2002 and is concurrent with Computer Science master degree.

He is the recipient of the Research Challenge Trust Fund (RCTF) fellowship and teaching assistantship at the University of Kentucky. His research interests include system modeling, discrete event simulation, temporal logic, formal verification, supervisory control, failure diagnosis, and applications of discrete-event systems.

Professional Publications

Journal

"Rules-based modeling of an Assembly Line and its Diagnosis", with V. Chandra and R. Kumar. *Asian Journal of Control*. Accepted 2003.

"Prioritized Composition with Exclusion and Generation for the Interaction and Control of Discrete Event Systems", with V. Chandra, W. Qiu and R. Kumar. *Mathematical and Computer Modeling of Dynamical Systems*. Accepted 2003.

"Modeling Discrete Event Systems with Faults using a Rules Based Modeling Formalism", with V. Chandra, S. Jiang and R. Kumar. *Mathematical and Computer Modeling of Dynamical Systems*. Accepted 2003.

"Automated Control Synthesis for an Assembly Line using Discrete Event System Control Theory", with V. Chandra and R. Kumar. *IEEE Transactions on Systems, Man, and Cybernetics, Part C*. May 2003, pp 33(2): 284-289.

"A polynomial algorithm for testing diagnosability of discrete event systems", with S. Jiang, Z. Huang and R. Kumar. *IEEE Transactions on Automatic Control*, August 2001, pp 46(8): 1318-1321.

Conference

"Modeling Discrete Event Systems with Faults using a Rules Based Modeling Formalism", with V. Chandra, S. Jiang and R. Kumar. *Proceedings of the 41th IEEE Conference on decision and control*, pages 4012-4017, Las Vega, NV, Dec 2002.

"Concurrent, asynchronous and generative interactions for the modeling and control of discrete event systems", With V. Chandra and R. Kumar, *Proceedings of 2003 American Control Conference*, pages 4010-4015, Denver, Colorado, June 2003.

"Diagnosis of Discrete Event Systems in Rule-based Model using symbolic analysis", With V. Chandra, S. Jiang and R. Kumar, accepted in Workshop on Model Checking and Artificial Intelligence (MoChArt-03), 2003.

Professional Affiliations

Student member, Institute of Electrical and Electronics Engineers (IEEE)

Zhongdong Huang

December 2, 2003