University of Kentucky
UKnowledge

University of Kentucky Doctoral Dissertations | Graduate School

2004

DIAGNOSIS OF CONDITION SYSTEMS

Jeffrey Ashley
University of Kentucky, ashtray@qx.net

Right click to open a feedback form in a new tab to let us know how this document benefits you.

Recommended Citation
Ashley, Jeffrey, "DIAGNOSIS OF CONDITION SYSTEMS" (2004). *University of Kentucky Doctoral Dissertations*. 341.
https://uknowledge.uky.edu/gradschool_diss/341

This Dissertation is brought to you for free and open access by the Graduate School at UKnowledge. It has been accepted for inclusion in University of Kentucky Doctoral Dissertations by an authorized administrator of UKnowledge. For more information, please contact UKnowledge@lsv.uky.edu.

ABSTRACT OF DISSERTATION

Jeffrey Ashley

The Graduate School

University of Kentucky

2004

DIAGNOSIS OF CONDITION SYSTEMS

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
Jeffrey Ashley

Lexington, Kentucky

Director: Dr. Lawrence Emory Holloway,
Professor of Electrical & Computer Engineering,
University of Kentucky,

Lexington, Kentucky

2004

Copyright © Jeffrey Ashley 2004

ABSTRACT OF DISSERTATION

DIAGNOSIS OF CONDITION SYSTEMS

In this dissertation, we explore the problem of fault detection and fault diagnosis for systems modeled as condition systems. A condition system is a Petri net based framework of components which interact with each other and the external environment through the use of condition signals. First, a system FAULT is defined as an observed behavior which does not correspond to any expected behavior, where the expected behavior is defined through condition system models. A DETECTION is the determination that the system is not behaving as expected according to the model of the system. A DIAGNOSIS of this fault localizes the subsystem that is the source of the discrepancy between output and expected observations. We characterize faults as a behavior relaxation of model components. We then show that detection and diagnosis can be determined in a finite number of calculations. The exact solution can be computationally involved, so we also present methods to perform a rapid detection and diagnosis. We have also included a chapter on a conversion from the condition system framework into a linear-time temporal logic(LTL) framework.

KEYWORDS: Discrete Event Systems, Condition Systems, Fault Diagnosis,
Fault Detection, Language relaxation.

Jeffrey Ashley

February – 20 – 2004

DIAGNOSIS OF CONDITION SYSTEMS

By

Jeffrey Ashley

Dr. Lawrence Emory Holloway
(Director of Dissertation)

Dr. William T. Smith
(Director of Graduate Studies)

February – 20 – 2004
(Date)

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 also requires the consent of the Dean of the Graduate School of the University of Kentucky.

DISSERTATION

Jeffrey Ashley

The Graduate School

University of Kentucky

2004

DIAGNOSIS OF CONDITION SYSTEMS

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
Jeffrey Ashley

Lexington, Kentucky

Director: Dr. Lawrence Emory Holloway,
Professor of Electrical & Computer Engineering,
University of Kentucky,

Lexington, Kentucky

2004

Copyright © Jeffrey Ashley 2004

ACKNOWLEDGMENTS

This work was supported, in part, by a National Science Foundation grant (ECS-0115694).

I would first like to thank Dr. Larry Holloway, my advisor, for his excellent leadership and mentorship over these last 10 years. I would also like to thank Dr. Ratnesh Kumar, my co-advisor, for his support and his insight over the last 9 years. I look forward to continued collaboration with the both of them in the future.

I would also like to thank my other committee members; Dr. Dietz, Dr. Yingling, and Dr. Brock for their advise and support. I also thank the Department of Electrical and Computer Engineering and the Center for Robotics and Manufacturing Systems at the University of Kentucky for previous financial support.

My chairperson at Kentucky State University, Dr. Ashok Kumar, has been very supportive these last few months. Thanks to his sensitivity, I have been able to finish my Ph.D. on time. Also thanks to my other coworkers for their recent support, notably Allison Noel, Mark Garrison, Jim Carrigan, Wassim Al-Hamdani, and Mike Unuakhalu.

Dr. Ram Potluri supplied the style and formatting files for this dissertation. His help made formatting this dissertation easy.

Finally, I would like to thank my mother, Phyllis Ashley, my family, Amy L. Leigh, and all of my other friends.

Thanks.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF FILES

ashley_diss.pdf	**876 KB**

Chapter 1

Introduction

This dissertation addresses the issue of failure detection and diagnosis for a class of modeling structures called condition systems. In this dissertation, we first establish the theoretical basis a fault detection and diagnosis using these models. We then present an exact solution for fault detection and diagnosis. This solution may be computationally cumbersome, so we present a method to integrate fault detection into our scheme for synthesizing controllers, and present a method to rapidly diagnose a system by exploiting the causal structure of our condition systems. We finish the contributions of this dissertation by showing how our system translates into the linear-time temporal logic framework.

In the next section we will frame the problem and briefly discuss current trends in this field of research. We finish this chapter with a brief discussion of our approach, the contributions of this work, and an overview of the rest of this dissertation.

1.1 Motivation

Within the context of control and supervision of discrete state systems, there are three distinct yet related problems: fault detection, fault diagnosis, and fault correction. Fault detection is the determination that the system behavior is not consistent with the defined allowed behavior. Fault diagnosis is localizing the fault. In other words, fault diagnosis is identifying subsystems for which their aberrant behavior could explain the observed aberrant behavior of the entire system. Fault correction is the process of modifying the system control to overcome a detected

and diagnosed fault. In this dissertation, we specifically focus on the fault detection and diagnosis task. These are well explored problems in computer science and in recent years have been the focus of much discrete event system (D.E.S.) research.

Faults can lead to down-time, customer frustration and loss, lost information, dangerous system behavior, and complete system failure that may lead to catastrophic accidents. Depending on the application, the system may be distributed over a large geographical area such as a telecommunications network, or it may require rapid diagnosis and correction for safety reasons such as faults found in electronics systems on military aircraft.

Computer science and computer engineering considers the problem of fault detection and diagnosis in many research fields because it is acknowledged as a persistently difficult problem[Dav84]. The most pertinent to this dissertation are process verification using temporal logic; model based reasoning; and truth maintenance systems.

The use of D.E.S techniques in diagnosis is rich and diverse, but generally all of the techniques use models to describe faulty behaviors, these are then used to create some sort of DIAGNOSER to determine a proper diagnosis given a set of observed outputs of the system (or a time ordered sequence of these sets). This is a problem because it complicates the modeling process [SSL+96]. Our approach varies in this respect because we do not require the modeling of faults (although we can model faults if required) which relieves the specifiers of a systems behavior of this potentially monumental task. We also note that in our work we distinguish between the detection problem and the diagnosis problem whereas in discrete event system literature they are treated together [CL01, SSL+95, SSL+96, JHCK01].

In our work, the models used to generate a diagnoser are generic models of the system behavior that do not necessarily require explicit modeling of faults. In fact, these models are principally used to generate control programs given a control objective among other things. A model represents the generic and unconstrained discrete behavior of some component of a system, and in their totality they represent the unconstrained yet normal operating behavior of a system independent of any control objective. Our models are typically (but not necessarily) primitive state

models that hopefully describe the general non-faulty open-loop behavior subsystem. The approaches found in the literature typically offer the advantage of solving more complicated problems though, as will be discussed in the next chapter. In the literature, multiple faults can occur, timed dynamics have been considered, and other assumptions embedded in the plant's modeling framework make much of this work generally more applicable.

1.2 The Approach

Condition system models are a form of Petri net for describing discrete-state systems composed of subsystems which interact through condition signals. The modular nature of these models allows a system modeler to describe the system through subsystems (or components) with well-defined interfaces. In [HGSA00], this modularity is exploited for control synthesis. Here, we show that this also makes condition systems well suited for detection and diagnosis. We note that all condition signals are not directly observable and this complicates the problem of detection and diagnosis. In this paper, we also assume that we encounter at most one faulty subsystem within our system.

To date, our approach toward diagnosis most closely follows the model-based reasoning approach of [DH88]. Specifically, we consider that we have a set of component models describing the expected behavior of each component and thus the complete system. A fault is an observed behavior that is no longer consistent with the modeled behaviors. Thus, we need not explicitly model faulty behaviors.

The work in this dissertation is based on the language behavior of condition systems. For fault detection and diagnosis, we have introduced the notion of the REAL and EXPECTED system. The expected system is a condition system model (specified by some human agent) of the plant that is used for synthesis and analysis. The real system is an abstraction of the actual system the expected system model represents. A fault detection is then defined as a language comparison between these two systems.

To define a detection and a diagnosis of a system, we introduce the notion of

a RELAXED LANGUAGE. In control synthesis we view subsystems in light of what is achievable, whereas in detection and diagnosis we view subsystems in terms of what constraints they impose on a system. A relaxed language then is the language generated by the system when the constraints of a subsystem are removed from consideration. If the language generated by an expected model of the plant, with some subsystem's constraints on its behavior removed, contains the sequence of outputs generated by the real plant then we know that this subsystem could account for the faulty behavior, and is hence a diagnosis.

We utilize the theoretical framework described above to: determine an exact solution to the detection and diagnosis problem (chapter 5); develop a controller based detector (based on the taskblock introduced in [HGSA00]) (chapter 6); develop a rapid diagnoser based on the work in causal networks by [DP94a] et. al.(chapter 7); and to show how condition systems translate into the linear-time temporal logic framework (chapter 8).

To show an exact solution to the fault detection and diagnosis problem, we start with the known observable state of the system at time zero, and calculate all possible legal markings (i.e. the state of our models) such that the observable state of the system does not change. From this we can calculate all legal next states with observations that differ from the initial observation. A detection and diagnosis are initiated whenever the observation from the system is illegal (i.e. not expected from our model). We continue with this for every different observation from the system. While this approach gives us a best solution within the constraints of observability it is computationally complex, especially when the marking space of our system models is large.

Our on-line fault detection scheme utilizes controller models called taskblocks. These models are guaranteed to drive the system to some target state under certain initial assumptions. In this dissertation, we show how to transform these taskblocks to detect unexpected responses from the system, and hence implement fault detection.

Our on-line fault diagnosis method allows us to rapidly determine an approximate diagnosis of a system by generating a diagnosis model that is based on sym-

bolic causal networks [DP94a], [DP94b] from our original condition system models. The dynamic condition system model describes state evolution and the evolution of condition signals over time, the diagnosis model abstracts this to describe the interrelationships between the components. In this way we can neglect past state information. We note that our method is inexact in that the diagnosis returned may include subsystems that cannot be the source of the fault, but it does in fact contain any subsystem that may be the source of the observed faulty behavior (i.e. this diagnosis method returns a superset of subsystem models that is guaranteed to contain the true diagnosis).

The final contribution to this work is a chapter that defines how we can convert a description of our system and a state trace of observations (called a condition sequence in this work) into the linear-time temporal logic framework. LTL is used in process verification which is closely related to the detection and diagnosis problem, and this framework is widely known among researchers in computer science and computer engineering. Our main goal of this chapter is to make our work more widely accessible to this community and to glean new insight into the detection and diagnosis problem.

The dissertation is organized as follows. In chapter 2 we discuss relevant research in the field of fault detection and diagnosis. We define condition systems and introduce the notion of relaxed languages in chapter 3. In chapter 4 we define the fault diagnosis problem within the context of languages of condition systems composed of multiple subsystems, and in chapter 5 we show that direct evaluation of our diagnosis definition is effectively computable (i.e. a solution can be found in a finite number of steps). This result leads us to an exact solution of the detection and diagnosis problem that can be computationally expensive. In contrast, we introduce causal networks in chapter 7 and show how the causal net can be used to rapidly generate a superset of diagnoses. While in chapter 6 we present a method to detect faults that complements the diagnosis method of chapter 7. In chapter 8, we show how our work relates to linear-time temporal logic. The dissertation concludes with a discussion on areas of future research.

Copyright © Jeffrey Ashley 2004

Chapter 2

Prior Research

Diagnosis of discrete systems is of interest to many researchers. It has practical implications in many fields including computer science, manufacturing, telecommunications, aerospace, and military applications. In this digital world, many large digital and discrete systems are being developed with a great deal of success but new issues have arisen. Among the forefront of these issues is the detection, diagnosis, and correction of faults in the system.

Fault detection, detection, and correction are important in the field of reliability engineering. Block diagram analysis and redundant system design are excellent examples of systematic approaches for designing reliability into complex systems. These ad-hoc methodologies have traditionally sufficed in complex system design (electronics, avionics, network design, etc.), but they typically add significantly to design, build and overhead cost. For an excellent reference in reliability engineering see [O'C91]. And oftentimes, manual diagnosis of such a system is difficult, time consuming, or impractical given the application. For these reasons, there is interest in the use of Discrete Event Systems (D.E.S.) techniques in detection, diagnosis and correction. D.E.S. and computer science research focuses on designed-in reliability as well as in on-line detection, diagnosis and correction.

This chapter highlights some relevant previous research by other investigators in this field. In particular, there is a review of diagnosis using D.E.S. techniques, and there is a review of diagnosis using model-based reasoning and model-based diagnostic techniques from the AI community.

In their book [CL01], Cassandras and Lafortune gracefully define diagnosis as "determining whether certain unobservable events COULD HAVE OCCURRED or HAVE

OCCURRED WITH CERTAINTY". The authors also give a formal method for developing a D.E.S. diagnoser which is a system that can perform a diagnosis. In their treatment a fault is considered to be an unobservable event; the plant, or system to be diagnosed, is represented as a non-deterministic automata; and a diagnoser is shown to be a modification of the deterministic automata representation of the plant (an OBSERVER). In [SSL+94, SSL+95], Sampath, Lafortune, et. al. introduce the notion of DIAGNOSABILITY for discrete event systems. Diagnosability defines whether a system can be diagnosed for all failures defined in the automata. In it, the authors exploit the cyclic nature of many discrete systems to determine conditions where unobservable events (faults) become uniquely identifiable (the diagnosis). In the system model as defined: failures are modeled within the state machine description of the plant; events leading to failures are unobservable; and failure event must be contained within cycles of the plant where at least one event is observable and distinguishes the failure. A nice feature is the allowance of multiple failures. A test for diagnosability is then shown that uses a diagnoser model which is exponentially complex in the number of states in the system.

Jiang, Kumar, et. al. present a method of testing diagnosability of a plant that is fourth order polynomial in its complexity [JHCK01]. They achieve this by eliminating the need to directly derive the diagnoser model in order to test diagnosability.

Sampath, Lafortune, et. al. apply their work to a HVAC system [SSL+96]. In it, they compare two systems, one that is diagnosable and the other that is not. Interestingly, the authors describe two crucial issues with their method. The first is in the complex nature of building the system model, and the authors describe the possibility of reuse of models for control, etc., much like in our approach. The second problem is with the computational complexity of the diagnostics process. These issues clearly impact scalability. Huang, Kumar, et.al. simplify the task of model construction by representing failures as rules [HCJK03] . A rules based modeling formalism is introduced to model failures. In using this technique, the complexity of the modeling of faults and failures goes from exponential in the number of faults to polynomial. The authors also consider timing in their work. Provan and Chen also consider timed systems [PC98, PC00]. In their work, they use causal networks

and a rules based formalism to allow for diagnosis of timed systems for a class of problems.

In [JKG03], the authors consider the detection and diagnosis problem for discrete systems where there may be repeated or intermittent failures. In this interesting work, the authors present a polynomial method to determine the number of times a fault has occurred. To this end they introduce the notion of K-DIAGNOSABILITY, and show under which conditions a system is K-diagnosable. In their approach, they use a transition graphs to test for K-diagnosability, and if the system passes these tests then a diagnoser (a state machine) can be created.

In [AFB+98], Aghasaryan et.al. present a modeling framework and approach to the diagnosis and detection problem using a variant of a one-safe stochastic Petri net. In this work, the models represent a distributed system, and a decentralized diagnoser is used for diagnosis. This work utilizes explicit fault models to determine where a potentially unobservable fault (system fault) has occurred based on the observations of observed alarms (events). A set of high level models describing the probabilistic and causal nature of faults and alarms within a system is utilized to generate a diagnoser. The diagnoser captures the expected sequential system behavior of the system and can be used for the diagnosis task. In this work, faults may lead to cascading faults, and alarm conditions may arrive to a diagnoser out of order with relation to their occurrence. This work is applicable to the analysis of telecommunication networks where this situation may occur. In this situation, the explicit modeling of faults is natural and required based on the complexity of the problem.

In [ZM99, ZKW99, ZKW03], the authors investigate diagnosis in the traditional DES framework. In [ZM99], necessary and sufficient conditions are presented for the generic solvability for fault detection and diagnosis of a system where simultaneous failures can occur. Interestingly, the authors have partitioned the problem into one of detection and diagnosis much like in our work.

In, [ZKW99], timed systems are considered for fault detection. In this work, a TICK event is used to represent timing changes, the system is described as a timed finite-state Moore automaton, and the diagnoser is a finite state machine. As in most

DES research, faults are described within the system model itself. The main contribution of this work, is a methodology to reduce the size of the diagnoser (which is potentially exponentially large due to incorporation of tick events). In [ZKW03], the consider the problem of model reduction (of the diagnoser) in the presence of partial observability with multiple faults possible. Faults are again assumed to be modeled within the system itself.

In [Had03], the author effectively presents a method for the detection of multiple faults in a non-concurrent manner for fully observable discrete state digital controllers. A fault as defined in this paper is actually a corruption of the controller state which can occur from noise for example. He utilizes redundancy to provide fault detection capabilities by adding states to the controller. His contribution is in the application of non-concurrent error detection which allows for a power-savings and higher circuit reliability. It is very applicable to the design of fault-tolerant digital controllers and observers.

Diagnosis is of interest within the Artificial Intelligence community because it is a good example of a reasoning problem. Researchers are interested in developing a consistent and workable theory of reasoning and the diagnosis problem serves as a good example for several reasons [Dav84]. The fact that it is of extreme practical interest is also a motivating factor.

In [Dav84], Davis presents a system that reasons about other systems using special programming languages to describe the structure and behavior of the system under consideration. The author considers the problem of component level troubleshooting of digital hardware in the example. The main thrust of the work, however, is motivational. The author frames the reasoning problem of which diagnosis is an example, and discusses many of the important concepts found in model-based reasoning. In this work, a system is composed of interacting components each of which can be modeled in some way. A failure occurs when some component(s) behaves in some non-expected way. Interestingly, the author also introduces the concept of CONSTRAINT SUSPENSION which is directly analogous to our idea of a relaxed language. The importance of causality is also discussed.

deKleer presents an expanded rules based reasoning systems called the as-

sumption based truth maintenance system (ATMS) in [deK86a, deK86b] . deKleer and Williams expand this work and apply it to the diagnosis of multiple faults in [dW87]. The systems the authors consider are also composed of component systems that interact in a well defined way. In it the authors introduce a diagnosis system called the general diagnostic engine (GDE). The GDE uses model based prediction with a modified ATMS system to diagnose systems. Probabilistic information about the system is derived automatically from the causal structure of the model and the probability information for the individual components. The GDE and the method presented is applied to the troubleshooting of digital circuits.

Davis and Hamscher also address the problem of diagnosis by using model-based reasoning techniques in [DH88]. In this work the description of the system does not include faulty behaviors. A fault is detected when the observed behavior is no longer consistent with the modeled behaviors. In [JdKR92], this is characterized as a MINIMAL DIAGNOSIS, because information about faults and exoneration strategies are not included. An exoneration strategy is a rule that excludes a component from certain diagnoses.

Struss and Dressler extend these ideas to include models of faulty behavior which results in a better diagnosis at the expense of modeling time [SD89]. The authors discuss how a-priori knowledge of typical failure modes properly represented can result in a dramatic improvement in the model-based reasoning approach to diagnosis. They use an interesting example, where the system is composed of a battery, three bulbs, and the interconnecting wires. In the model based reasoning approach using the GDE of deKleer[dW87], a diagnosis of a system with two light bulbs non-functioning could result in a diagnosis where the battery is dead and a wire is incorrectly supplying voltage! Addition of failure information can eliminate these types of obviously erroneous diagnoses.

Symbolic causal networks [DP94a, DP94b, PC98] provide a systematic way of constructing logically correct databases used in reasoning systems. Symbolic causal networks are directed acyclic graphs used to describe causal influences, and by the method of construction they insure that causal inconsistencies are avoided. This is another example of the use of model-based reasoning techniques. Timing can

be incorporated in these models, and in [PC98] the authors utilize temporal causal networks to consider fault detection in the DES framework.

Temporal logic is of great interest to the computer science community and there is a host of literature covering different flavors of temporal logic, a few of which are CTL, CTL*, and LTL. As stated, temporal logic is largely used in addressing the verification problem. A Kripke structure (a labeled finite state machine) specifies some process, and temporal logic statements can be used to specify desired properties of this process. In verification, this information is used to determine whether the specified properties are satisfied by the Kripke structure. For some good references in temporal logic see [HR00, MP91, Kro87, GO94]. Temporal logic has also been applied to problems in DES. In [JK03c], temporal logic is used for the problem of supervision. In [JK03b, JK03a], the authors consider the use of temporal logic for the problem of diagnosis.

In the next chapter, we review condition system models and the languages generated by these models. We then introduce the notion of relaxed languages.

Copyright © Jeffrey Ashley 2004

Chapter 3

Condition Systems

In this dissertation, we consider systems represented by CONDITION SYSTEMS. Condition systems are a form of Petri net with explicit inputs and outputs called CONDITIONS. These conditions allow us to represent the interaction of subsystems, as well as the interaction of a system with a controller [HGSA00]. The condition system models share similarities to interpreted Petri nets. They are a subset of the condition-event models developed by Sreenivas and Krogh, and considered by Hanisch, Kowalewski, and others [HLR97, SK91, KLLU98].

Our main interest in the use of Condition System models in this work is twofold. First, condition system models that describe the open loop behavior of a system are already being used in control synthesis among other things, and our interest is in the reuse of these models. The generation of models is problematic (which hopefully condition systems alleviate to some degree) and the reuse of models could result in a huge time savings in system design. It would also help insure that a controller and a diagnoser are more consistent.

The second reason for use of condition system models over traditional DES models deals with how cause and effect relationships are clearly defined in a condition system model. Many discrete systems have a well defined cause and effect relationship between subsystems. Condition systems (under our assumptions) capture this gracefully by the use of state communication and by the use of a partition on the set of conditions into input and output conditions for some subsystem. With condition systems, descriptions of a real world system can be modeled using a more direct cause and effect methodology. The models also clearly show these causal relationships.

This chapter is organized into two sections. In the first section, we review condition systems, present our first assumption, and introduce the examples used in the remainder of the dissertation. In the second section, we focus on the language generated by a condition system. Here we introduce the notion of a relaxed language, and we define a failure and failure diagnosis in terms of relaxed languages.

3.1 Condition System Models

In this section, we define a condition system as a form of Petri Net that requires conditions for enabling of transitions, and that outputs conditions (establishes the truth of certain conditions) according to its marking. Further discussion of the condition systems we consider can be found in [HGSA00, HA98a].

A condition system is composed of distinct subsystems (each is represented by a model). The systems that we consider interact with each other and with their outside environment through CONDITIONS. A condition is a signal that either has value "true", or "false".

Let *AllC* be the universe of all conditions, such that for each condition c in *AllC*, there also exists a negated condition denoted $\neg c$, where $\neg(\neg c) = c$.

Definition 3.1 formally defines condition systems that we consider for this dissertation. Refer to figure 3.1 for an example condition system.

Definition 3.1 A condition system $\mathcal{G}$ is characterized by a finite set of states $M_{\mathcal{G}}$, a next state mapping $f_{\mathcal{G}} : M_{\mathcal{G}} \times 2^{AllC} \longrightarrow 2^{M_{\mathcal{G}}}$, and a condition output mapping $g_{\mathcal{G}} : M_{\mathcal{G}} \longrightarrow 2^{AllC}$. In this dissertation, we assume that $M_{\mathcal{G}}$, $f_{\mathcal{G}}$, and $g_{\mathcal{G}}$ are defined through a form of Petri net consisting of a set of places $\mathcal{P}_{\mathcal{G}}$, a set of transitions $\mathcal{T}_{\mathcal{G}}$, a set of directed arcs $\mathcal{A}_{\mathcal{G}}$ between places and transitions, and a condition mapping function $\Phi_{\mathcal{G}}(\cdot)$, where $(\forall p)\Phi_{\mathcal{G}}(p) \subseteq AllC$ maps output conditions to each place, and $(\forall t)\Phi_{\mathcal{G}}(t) \subseteq AllC$ maps ENABLING CONDITIONS to each transition. The net is related to $M_{\mathcal{G}}$, $f_{\mathcal{G}}$, and $g_{\mathcal{G}}$ in the following manner:

1. THE STATES ARE THE MARKINGS OF THE PETRI NET: each state $m \in M_{\mathcal{G}}$ is a function over $\mathcal{P}_{\mathcal{G}}$ that represents a mapping of non-negative integers to places.

2. THE OUTPUT CONDITIONS HAVE THEIR TRUTH VALUE ESTABLISHED BY MARKED PLACES: for any $m \in M_{\mathcal{G}}$,
   $g_{\mathcal{G}}(m) = \{c \mid \exists p \text{ s.t. } c \in \Phi_{\mathcal{G}}(p) \text{ and } m(p) \geq 1\}$, where $g_{\mathcal{G}}(m)$ is the set of output conditions forced "true" by marking m.

3. NEXT-STATE DYNAMICS DEPEND ON STATE ENABLING AND CONDITION ENABLING: for any $m \in M_{\mathcal{G}}$ and any set $TrueC \subseteq \mathit{AllC}$ of conditions with value "true", $m' \in f_{\mathcal{G}}(m, TrueC)$ if and only if there exists some transition set T such that

   (a) T is STATE-ENABLED, meaning $(\forall p \in \mathcal{P}_{\mathcal{G}})\ m(p) \geq |\{t \in T \mid p \text{ is input to } t\}|$

   (b) T is CONDITION-ENABLED, meaning $(\forall t \in T)\ \Phi_{\mathcal{G}}(t) \subseteq TrueC$

   (c) the next marking m' satisfies $\forall p \in \mathcal{P}_{\mathcal{G}},\ m'(p) = m(p) - |\{t \in T \mid p \text{ is input to } t\}| + |\{t \in T \mid p \text{ is output of } t\}|$

4. $M_{\mathcal{G}}$ IS CLOSED UNDER $f_{\mathcal{G}}(\cdot)$: if $m \in M_{\mathcal{G}}$ and $m' \in f_{\mathcal{G}}(m, TrueC)$ for some $TrueC \subseteq \mathit{AllC}$, then $m' \in M_{\mathcal{G}}$.

We note that items in 3a and 3c above correspond to standard Petri net state enabling and firing of a transition set, respectively. Item 3b adds an additional transition set enabling constraint that the input conditions to each transition must also be within the considered set $TrueC$ of true conditions.

Figure 3.1 shows an example condition system. As in standard Petri Net literature, places are represented by circles, transitions are represented by bars, directed arcs show connectivity, and the state of the system is represented by tokens located within places. Dashed arcs indicate the flow of conditions between subsystems.

For this dissertation, we have also refined our definition of a condition system by requiring that the set of markings, $M_{\mathcal{G}}$, be finite. We believe that this restriction is entirely realistic, and will limit the modeling power of the system very little in regards to our aims.

We are principally interested in using these models first to describe the open loop behavior of a system, and second to describe specifications of expected behavior which are used for controller generation. It is generally true that real world systems are typically limited in capability by physical constraints, and in as much

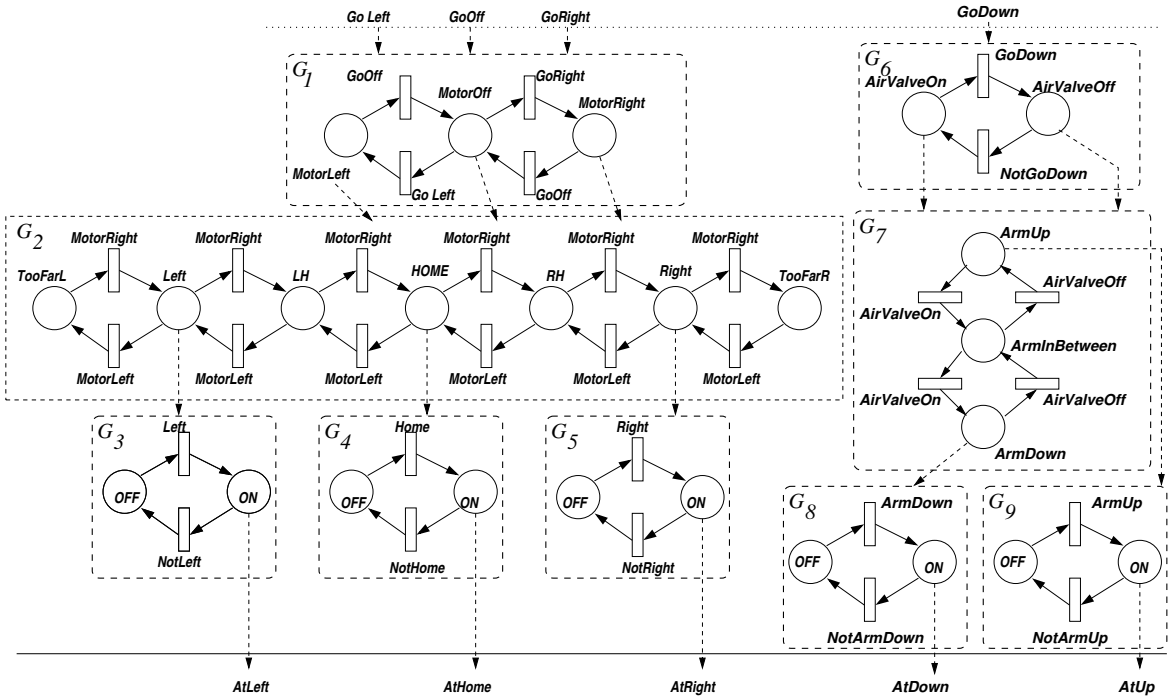

Figure 3.1: The Condition System model of the Plant (Part 1-Pick and Place System).

are amenable to modeling using models that are bounded in the number of tokens they can have. Likewise, specifications are typically limited in scope and are also amenable to modeling using such models.

We define the output condition set for a system $\mathcal{G}$ as $C_{out}(\mathcal{G}) = \{c \in \Phi_{\mathcal{G}}(p) \mid p \in \mathcal{P}_G\}$. Similarly, define $C_{in}(\mathcal{G}) = \{c \in \Phi_{\mathcal{G}}(t) \mid t \in \mathcal{T}_{\mathcal{G}}\}$. An example of a condition system is shown in figure 3.1.

Note that a condition system can be subdivided into subsystems, where each subsystem is a condition system over a set of connected places and transitions which are disconnected from all other places and transitions. In the example of figure 3.1, there are eight such subsystems. For the remainder of this dissertation, we use the notation $\mathcal{G}$ to indicate the complete system, and the notation $\{G_1, \ldots G_n\}$ to indicate the set of subsystems in $\mathcal{G}$. Given an initial marking m_0 of $\mathcal{G}$, we let $m_{0,i}$ denote the marking over just the places in $G_i \in \mathcal{G}$. We also make the following assumption on

the structure of our subsystems.

Unique Condition Source Assumption (UCS Assumption): For any subsystem $G_i \in \mathcal{G}$ and any condition $c \in C_{out}(G_i)$, the following are assumed to hold:

1. c IS NOT AN OUTPUT OF ANY OTHER SUBSYSTEM: $\forall j \neq i$, $c \notin C_{out}(G_j)$.

2. G_i DOES NOT OUTPUT CONTRADICTIONS: the condition system G_i is such that for all markings $m \in M_{G_i}$, either $c \in g_{G_i}(m)$ or $\neg c \in g_{G_i}(m)$, but not both.

Item 1 ensures the modularity of the system by requiring that each condition is output by at most one subsystem, G_i. This is a natural assumption for the types of systems we consider. Namely, we are interested in describing discrete systems that are developed, designed and implemented in a distributed and hierarchical manner. Typically, interactions between elements in such a system are required and are naturally unique. Due to item 2 of the assumption above, we will simplify our examples by only labeling places which output the positive value of a given condition c, and we omit the explicit labeling of places of the negation $\neg c$. We similarly will omit the negation of conditions when discussing condition sets, when the meaning should be clear. [1]

In previous work, [HGSA00],[GH00],[GH01], we require that the open loop model describing the system be bound to one token per subsystem. In this dissertation, we allow multiple tokens per subsystem, and so the system described above actually extends the model capabilities considered.

Example 3.1 Consider a simple assembly cell shown in figure 3.2. Sub-assemblies are loaded onto a conveyor system under certain restrictions. An assembly machine places a mating part onto the sub-assembly and the conveyor delivers the finished part to the part chute.

A portion of the condition system model of the assembly station is shown in figure 3.1. The rest of the model is shown in figure 3.6 and is not used in this

[1] We note that although the UCS assumption is logical for physical system models, the first statement of the assumption does not apply to the CONTROLLER LOGIC condition system models as synthesized by the methods in [HGSA00].

example. The assembly station takes parts from the parts bin and places them onto a waiting assembly. Our approach is to generate a plant model that captures the open-loop behavior (with a few exceptions included for brevity) of a system. Inputs to the top level models represent actuation signals that drive the system to new outputs. The outputs located at the bottom of the figures represents the output conditions of a plant, and typically represent sensors and other observations from the plant.

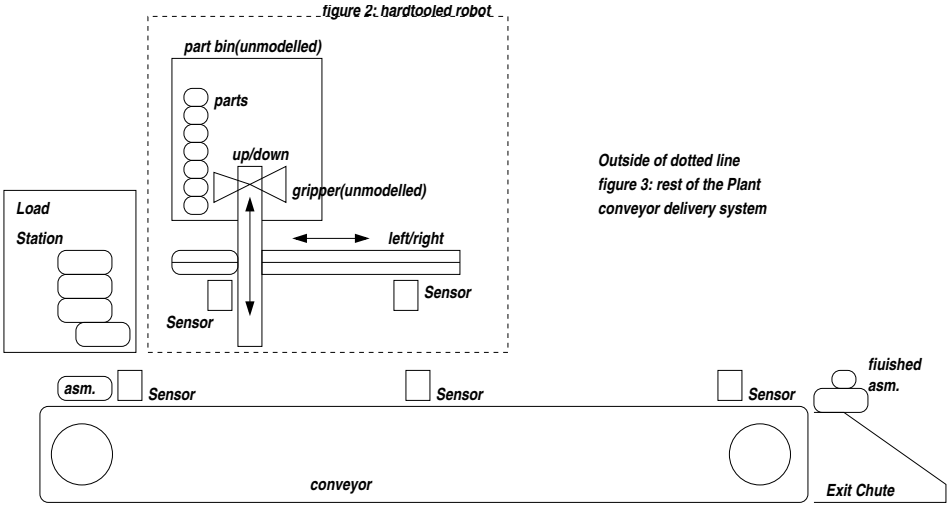

Figure 3.2: A depiction of the system used in the examples for this dissertation.

3.2 Condition system languages and the descriptive ordering

In the section, we introduce the notation and formalisms of the language generated by condition systems. We also introduce the notion of a descriptive ordering which will allow us to compare elements of these languages (sequences of condition sets). Elements of this language are sequences of condition sets that can represent the evolution of a condition system. These sequences can also be used to specify the desired behavior of some system (i.e. a specification mechanism). In this dis-

sertation we are also interested in using these ideas in the definition of a relaxed language, a fault detection and a fault diagnosis.

The behavior of a condition system can be described by sequences of condition sets. A condition set sequence, called a C-SEQUENCE, is a finite length sequence of condition sets. Each condition set sequence is of the form $(C_0C_1 \dots C_k)$ for some integer k and sets $C_i \subseteq \mathit{AllC}$ for all $0 \leq i \leq k$. The sequence is said to be CONTRADICTION FREE if for each C_i for any $c \in C_i$, then $\neg c \notin C_i$. A set of C-sequences is called a language, and the language consisting of all C-sequences is denoted $\mathcal{L}$.

Definition 3.2 Given a condition system $\mathcal{G}$ and a marking m_0, define the language $L(\mathcal{G}, m_0) \subseteq \mathcal{L}$ to be the set of condition set sequences such that $(C_0C_1C_2 \dots C_n) \in L(\mathcal{G}, m_0)$ if there exists some marking sequence $m_0m_1 \dots m_k$ and index mapping function $j(i)$ with $j(0) = 0$, $j(k) = n$ such that

1. MARKINGS EVOLVE ACCORDING TO CONDITIONS:
   $m_{i+1} \in f_{\mathcal{G}}(m_i, C_{j(i)})$ for $0 \leq i \leq n-1$.

2. OUTPUT CONDITIONS RESULT ONLY FROM THE MARKING:
   $g_{\mathcal{G}}(m_i) = C_{j(i)} \cap C_{out}(\mathcal{G})$,

3. SEQUENCING IS MAINTAINED: A marking m_{i+1} either maps to condition sequence element $C_{j(i)}$ corresponding to prior marking m_i in the marking sequence, or it maps to the next condition sequence element. More formally, for any $0 \leq i < k$,
   $j(i+1) = j(i)$ or $j(i+1) = j(i) + 1$;

The above definition deserves some explanation. The notation $C_{j(i)}$ indicates the condition set associated with the ith marking. By statement 2, the output conditions in set $C_{j(i)}$ correspond to the marking m_i, and by statement 1, marking m_i evolves to marking m_{i+1} only if enabled under condition set $C_{j(i)}$.

The marking sequence and condition set sequence have different indices because the mapping between the sequences is not necessarily one-to-one. For example, a marking could change from m_i to m_{i+1}, but $g(m_i)$ and $g(m_{i+1})$ could be the same. Thus, it is possible that both markings could correspond to the same condition set

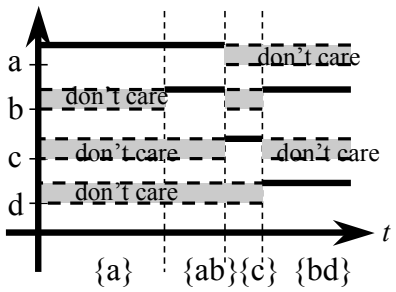

Figure 3.3: Examples of signal time lines corresponding to the C-sequence $s = (\{a\},\{ab\},\{c\},\{bd\})$.

in the given C-sequence. This then implies that there could be fewer condition sets in the C-sequence than distinct markings in the corresponding marking sequence.

Furthermore, note that for any m and any C, $m \in f_{\mathcal{G}}(m, C)$, since it is possible that no transitions fired. From statement 3, then it is possible that $m_{i+1} = m_i$. These two identical markings could map to different condition sets, so it is possible to have more condition sets in the C-sequence than distinct markings in the corresponding marking sequence. Finally, we point out that $L(\mathcal{G}, m_0)$ is obviously prefix-closed (excluding the empty prefix).

A C-sequence can be viewed as a sequence of conditions that must be true over certain unspecified though ordered time periods. Given a C-sequence $s = (C_0 C_1 \ldots C_n)$ and some $0 \leq i \leq n$, C_i represents a subset of conditions (or negated conditions) that are true for some (possibly non-unique) period of time. C_i does not have to include all true conditions over the time period. However, the time period that C_i represents must follow immediately after the time period represented by C_{i-1}, and must be followed immediately by the time period represented by C_{i+1}. Thus, for the sequence $s = (\{a\}\{ab\}\{c\}\{bd\})$, one such time-line is shown in figure 3.3. Note that the condition b may be true throughout the time line, but does not have to be listed in all condition sets in the sequence. This is analogous to a "don't care" condition on its value when it is not specified.

We need to describe important characteristics of condition sequences without

listing all details of all condition activity within the sequence. A convenient way to characterize a sequence is through a partial ordering "$\leq$" which we refer to as the DESCRIPTIVE ORDERING, which can be used to compare features of different sequences. Further explanation of the ordering is given in [HA98b]. We define it again below: [2]

Definition 3.3 Define the DESCRIPTIVE ORDERING $\leq$ over condition sequences such that

1. $(C_1 C_1') \leq (C_2)$ if $C_1 \subseteq C_2$ and $C_1' \subseteq C_2$.
2. $(C_1) \leq (C_2 C_2')$ if $C_1 \subseteq C_2$ and $C_1 \subseteq C_2'$.
3. Given C-sequences s_1, s_1', s_2, and s_2' such that $s_1 \leq s_1'$ and $s_2 \leq s_2'$, then $s_1 s_2 \leq s_1' s_2'$
4. If $s_1 \leq s_2$ and $s_2 \leq s_3$, then $s_1 \leq s_3$.

From the definition above, we see that given sequences s_1 and s_2, if $s_1 \leq s_2$, then s_1 contains no more condition information in it than s_2, and s_2 can be said to be AT LEAST AS DESCRIPTIVE as s_1. If $s_1 \leq s_2$ and $s_2 \leq s_1$, then the sequences are said to be equivalent under the ordering, written as $s_1 \equiv s_2$.

Items 1 and 2 in the definition above establish the ordering based on subsets of condition sets, item 3 considers the concatenation of smaller ordered sequences, and item 4 defines the ordering to be transitive. Recall that conditions that are not listed are either "don't care" or "don't know" conditions. The descriptive ordering lets us omit consideration of specific conditions during periods when they are not of interest, while still allowing comparison of some basic sequencing characteristics.

Example 3.2 To illustrate the descriptive ordering, consider the following condi-

[2] In earlier works such as [HA98b], the term ELABORATIVE ORDERING was used instead of descriptive ordering.

tions.

$$
\begin{aligned}
s_1 &= (\{a\}\{b\}\{a\}) \\
s_2 &= (\{a\}\{a\}\{a\}\{b\}\{a\}) \\
s_3 &= (\{a, c\}\{b, c\}\{a, d\}) \\
s_4 &= (\{a, b\})
\end{aligned}
$$

The following relationships are true.

$$s_1 \equiv s_2 < s_3$$

$$s_1 \equiv s_2 < s_4$$

However, s_3 and s_4 are not comparable under the descriptive ordering. To illustrate why, note that $\{a, b\} \not\subseteq \{a, c\}$ and $\{a, c\} \not\subseteq \{a, b\}$, so the sequences cannot be ordered.

Let $(AllC)$ be the C-sequence of length one that consists of all conditions (including negations). Note that it is inherently contradictory. Let $(\emptyset)$ be the C-sequence of length one that consists of no conditions. The following results can be shown.

Lemma 3.1 The following statements are true:

1. $s \leq (AllC)$ for any C-sequence s ;
2. $(\emptyset) \leq s$ for any C-sequence s ;
3. $(C) \equiv (CC) \equiv (CCC) \equiv (C^n)$ for any condition set C and any $n > 0$;
4. Given C-sequences s_1 and s_2 and condition set C, $s_1 C s_2 \equiv s_1 C C s_2$;
5. Given C-sequences s_1 and s_2, and condition sets C, C', if $C \subseteq C'$, then $s_1 C s_2 \leq s_1 C' s_2$.

Note that statement 1 of lemma 3.1 says that the condition sequence consisting of the set of all conditions (and their negations) being true is the most descriptive of all C-sequences (but it is contradictory). Statement 2 says that the sequence consisting of just an empty set of conditions is the least descriptive C-sequence,

since it says nothing about the truth value of any condition at any time. Statement 3 says that any finite nonzero length sequence of a condition set is equivalent to any other finite nonzero length sequence of the same set. Statement 4 says that duplication of a condition set within a sequence results in an equivalent sequence. Statement 5 considers two sequences that differ only in a single condition set, where the set in the first sequence is a subset of the set in the second sequence. The statement then says that the second sequence is more descriptive.

Proof of lemma 3.1:

(From [HA98a]) We prove the lemma statements in reverse order. To show lemma statement 5, note that by definition 3.3 items 1 and 2, $(C) \leq (CC) \leq (C')$, so by transitivity, $(C) \leq (C')$. Concatenating these onto s_1 and then concatenating on s_2, using the definition's item 3 then gives us the lemma's statement 5.

Statement 4 says that insertion of a duplicate condition in a sequence results in an equivalent C-sequence. From definition 3.3 items 1 and 2, it follows that $(CC) \equiv (C)$. Statement 4 then follows with the definition item 3 by concatenating these onto the sequence s_1 and then concatenating on s_2.

Lemma statement 3 is a direct consequence of the repeated application of statement 4 on a sequence of identical condition sets. To show statement 1, substitute each individual condition set in the sequence s with the set *AllC*. By the lemma item 5, this sequence of repeated *AllC* is more descriptive than the original sequence, and by statement 3, it is equivalent to the unit-length sequence $(AllC)$. Lemma statement 2 is done in a similar manner. □□□

3.3 Relaxed Languages

In this section, we introduce the concept of a RELAXED language which will be used in the next chapter to define a diagnosis of a condition system. Briefly a relaxed language is the language generated by a system when the constraints imposed by a subsystem are removed. The notion of relaxed language is used in the identification of faulty subsystems. We conclude this section with some examples of the concepts

introduced in this section.

It is important to note from Definition 3.2 that for a subsystem G_i, except for output conditions of G_i and for input conditions required for transition enabling in G_i, whether some other condition is true or false or unspecified does not influence a C-sequence's inclusion in $L(G_i, m_{0,i})$. Thus, $L(G_i, m_{0,i})$ by its definition will contain C-sequences with all possible valuations over time of such "irrelevant" conditions. This observation leads us to our first main contribution of this dissertation. In this first proof, we show that the language generated by the system is the intersection of the subsystem languages. This will allow us to establish the relaxed language framework on which this research is based.

Theorem 3.1 Given a condition system $\mathcal{G} = \{G_1 \dots G_n\}$ satisfying the UCS assumption, then:

$$L(\mathcal{G}, m_0) = \bigcap_{1 \le i \le n} L(G_i, m_{0,i}) \tag{3.1}$$

Proof: Follows directly from lemma 4 of [HGSA00]. By Definition 3.2, any C-sequence s in the language $L(\mathcal{G}, m_0)$ must correspond to a sequence of markings of $\mathcal{G}$. Projecting these markings only on the places in a subsystem G_i, we see that the C-sequence also must correspond to this sequence of markings over G_i, so $s \in L(G_i, m_{0,i})$ also. To prove in the other direction, it is sufficient to note that for some $s \notin L(\mathcal{G}, m_0)$, some statement in definition 3.2 must be violated. Statement 1 just defines an indexing relationship. Suppose that such an indexing is possible up to condition set C_k in the sequence, but no such indexing is possible thereafter due to either the violation of statement 2 or statement 3. If it is statement 3 that is violated, then some transition necessary for the progression of markings must not be enabled by the condition sequence. This would also violate the transition enabling for the subsystem G_i containing that transition, so $s \notin L(G_i, m_{0,i})$ also. For statement 2 to be violated, the first case is that some condition that is output from a marked place is not shown in the corresponding condition set. This would mean that for the subsystem G_i containing that marked place, $s \notin L(G_i, m_{0,i})$. The second case is that some output condition is in C_{k+1}, but there is not a corresponding

marking. By the UCS assumption, the condition is uniquely associated with some subsystem G_i, so it is also true that $s \notin L(G_i, m_{0,i})$. We have thus shown the theorem. □□□

Extending this idea, we define the language created by neglecting the constraints imposed by one subsystem. We call this a RELAXED language since the language is larger than and contains the original language. To generate the new subsystem $G_{i,NEW}$ that, when combined with the other subsystem models, implements the relaxed behavior we merely need to create two places for each output condition in the original subsystem ($\forall c \in C_{out}(G_i)$). One place outputs c and one place outputs the negated condition $\neg c$ (in the figure: $\Phi_{G_{i,new}}(p_1) = \{c\}$ and $\Phi_{G_{i,new}}(p_2) = \{\neg c\}$. Connect these places with two transitions that have no enabling conditions (i.e. $\Phi_{G_{i,new}}(t) = \emptyset$ for both t) as shown in figure 3.4

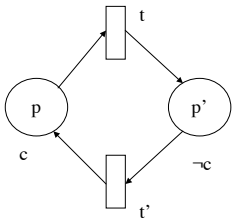

Figure 3.4: Generating the portion of the net that outputs c or $\neg c$

We assign one token for each pair of these places. We pick an initial marking $m_{0,i,NEW}$ for the new subsystem such that the requirement $g(m_{0,i}) = g(m_{0,i,NEW})$ holds. This obviously can be accomplished in a finite number of steps. An example is shown in figure 3.5

Note that $L(G_{i,NEW}, m_{0,i,NEW})$ is the set of all contradiction free sequences for which the beginning condition set observation is consistent with the observation expected from the given initial marking. THIS IS INDEPENDENT OF ANY NET DYNAMICS, so for any sequence in $L(G_{i,NEW}, m_{0,i,NEW})$, all condition sets after the first are entirely unrestrained (other than being contradiction free).

Definition 3.4 Given a system $\mathcal{G}$ and some $G_i \in \mathcal{G}$, define the language $L(\mathcal{G}/G_i, m_0)$

as

$$L(\mathcal{G}/G_i, m_0) := \left(\bigcap_{1 \le j \le n, j \neq i} L(G_j, m_{0,j}) \right) \cap L(G_{i,NEW}, m_{0,i,NEW}) \tag{3.2}$$

From Theorem 3.1, we see that we can view each subsystem $G_i \in \mathcal{G}$ as a constraint that restricts the behavior of conditions in $C_{out}(G_i)$ within $L(\mathcal{G}, m_0)$. We can thus view $L(\mathcal{G}/G_i, m_0)$ as a relaxed language, because we are thus removing the constraint (other than initial condition values) that G_i imposed on the language. Note that it follows from the definition that $L(\mathcal{G}, m_0) \subseteq L(\mathcal{G}/G_i, m_0)$ for any $G_i \in \mathcal{G}$.

Additionally, we will say the system $\mathcal{G}$ is RELAXED WITH RESPECT TO a subsystem G_i to identify the system with language $L(\mathcal{G}/G_i, m_0)$.

Finally we conclude this section with a brief definition of observability over conditions. Let $C_{obs} \subseteq \mathit{AllC}$ be a set of conditions which can be observed, where it is implied that if $c \in C_{obs}$, then $\neg c \in C_{obs}$. These observed conditions can include inputs and outputs of $\mathcal{G}$, and potentially conditions that have no relationship to $\mathcal{G}$. Given a C-sequence $(C_0 C_1 \dots C_k)$, define the projection over some set $C_{obs} \subseteq \mathit{AllC}$ as

$$(C_0 C_1 \dots C_k) \mid_{C_{obs}} := ((C_0 \cap C_{obs})(C_1 \cap C_{obs}) \dots (C_k \cap C_{obs}))$$

This is generalized over sets of C-sequences using our equivalence relationship for C-sequences. Thus, the projection of the language over the observable conditions is defined as:

$$L(\mathcal{G}, m_0) \mid_{C_{obs}} := \{s \mid \exists s' \in L(\mathcal{G}, m_0) \text{ s.t. } s \equiv s' \mid_{C_{obs}}\}$$

Example 3.3 Suppose that

$$s' = (\{c_1, c_2\}\{c_1, \neg c_2\}\{\neg c_1, \neg c_2\}\{\neg c_1, c_2\}) \in L(\mathcal{G}, m_0)$$

If $C_{obs} = \{c_1, \neg c_1\}$, then

$$s' \mid_{C_{obs}} = (\{c_1\}\{c_1\}\{\neg c_1\}\{\neg c_1\})$$

and is in $L(\mathcal{G}, m_0) \mid_{C_{obs}}$. Since the sequence $(\{c_1\}\{\neg c_1\}) \equiv (\{c_1\}\{c_1\}\{\neg c_1\}\{\neg c_1\})$, then also

$$(\{c_1\}\{\neg c_1\}) \in L(\mathcal{G}, m_0) \mid_{C_{obs}}$$

Example 3.4 Consider a plant represented by figure 3.1. There is a horizontal motion actuator and a vertical motion actuator. There are 9 subsystems in this plant including 5 subsystems that input or output observable conditions, $(G_1, G_3, G_4, G_5, G_6, G_8, G_9)$. The condition inputs to G_1 and G_6 are considered control inputs to the plant and are hence observable. The condition outputs of G_3, G_4, G_5, G_8, and G_9 are sensor signals, and are thus also observable. The set of C_{obs} of observable conditions consists of conditions GoLeft, GoRight, GoOff, GoDown, AtLeft, AtHome, AtRight, AtDown, AtUp and their negations.

An example C-sequence representing movement of the horizontal position from Left to Home (neglecting vertical motion subsystems $G_6 - G_9$ and omitting negated conditions) would be:

$$\begin{aligned} s' = (&\{\text{Left}, \text{AtLeft}, \text{GoRight}, \text{MotorRight}\},\\ &\{\text{LH}, \text{AtLeft}, \text{GoRight}, \text{MotorRight}\},\\ &\{\text{LH}, \text{GoRight}, \text{MotorRight}\},\\ &\{\text{Home}, \text{GoRight}, \text{MotorRight}\},\\ &\{\text{Home}, \text{AtHome}, \text{GoRight}, \text{MotorRight}\}) \end{aligned}$$

(We omit the negative conditions for ease of discussion. However, by the UCS Assumption, if AtHome is not listed as explicitly true but is an output condition of the system being considered, then the negated condition $\neg$AtHome is assumed true.) Note that we pass through the outputs $\{\text{LH}, \text{AtLeft}, \text{GoRight}, \text{MotorRight}\}$ and $\{\text{Home}, \text{GoRight}, \text{MotorRight}\}$ briefly, assuming that the sensors respond to the arm positions very quickly.

The observable C-sequence from s' is then

$$\begin{aligned} s'|_{C_{obs}} = (&\{\text{AtLeft}, \text{GoRight}\},\\ &\{\text{GoRight}\},\\ &\{\text{AtHome}, \text{GoRight}\}). \end{aligned}$$

To complete this chapter, we include the condition system models as shown in figure 3.6. These sub-assemblies complete the simple assembly cell of figure 3.2. This model captures the behavior of the of the conveyor delivery system and loading operation.

Copyright © Jeffrey Ashley 2004

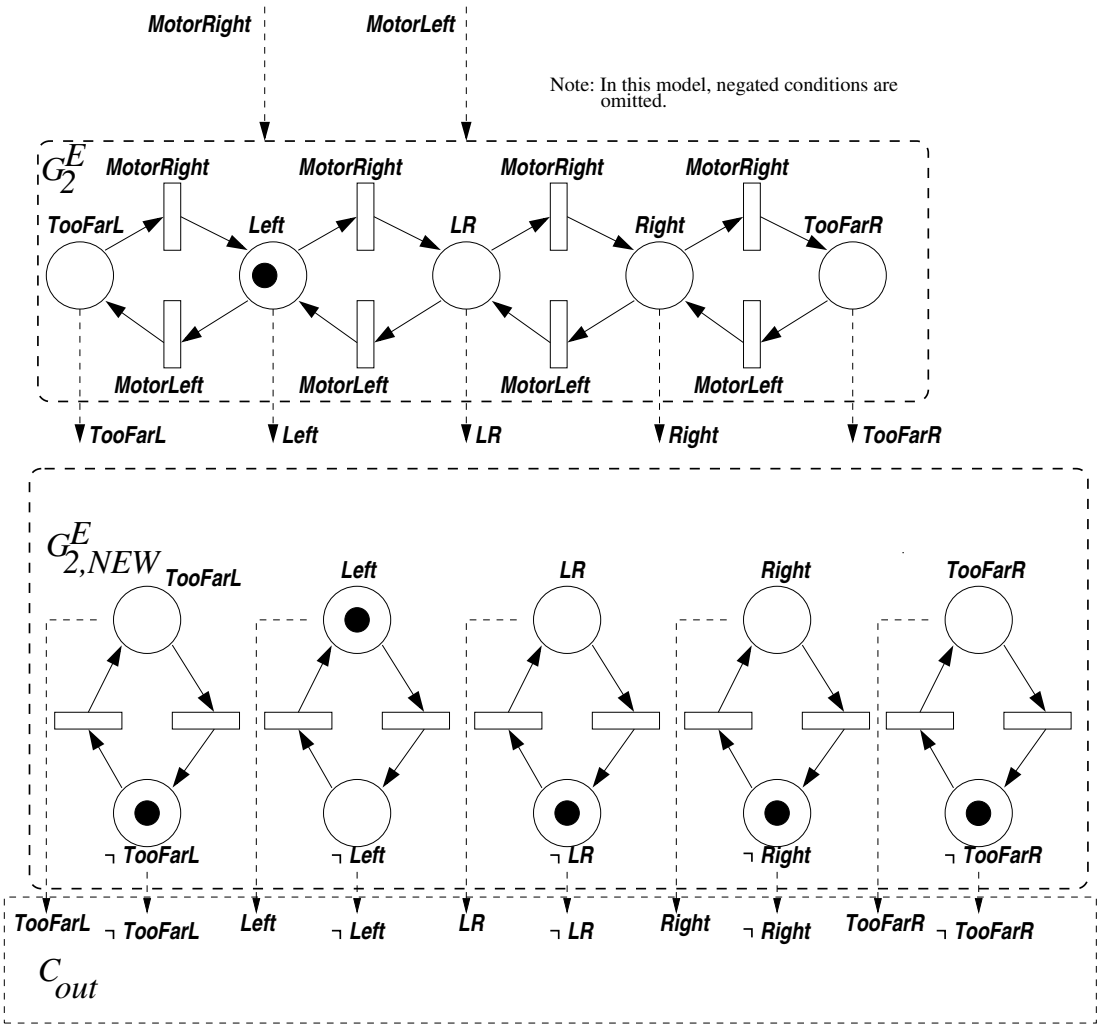

Figure 3.5: An example of creating $G_{i,NEW}$ from G_i

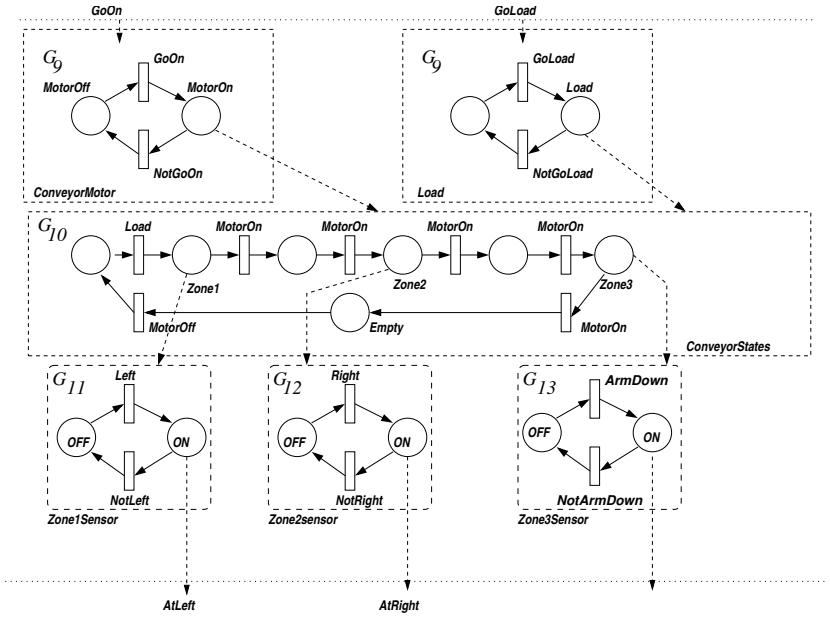

Figure 3.6: The Condition System model of the Plant (Part2-Conveyor Delivery System).

Chapter 4

Fault Diagnosis in Condition Systems

In this chapter, we formally define the detection and diagnosis problem within the condition system framework. Briefly, a detection operates on an observed sequence and when the observed sequence from the plant does not match any of the expected behaviors within the model of the plant then we have a fault. The diagnosis returns a set of potential candidate components that could be the source of the problem.

For a given system, we use superscripts to distinguish between the "real" system, $\mathcal{G}^R$, and our "model" system of expected behavior, $\mathcal{G}^E$. We also extend the superscript to the C-sequences and markings that we consider. For example, a C-sequence of the "real" system is denoted by $(C_0^R \dots C_{k-1}^R C_k^R)$ and the initial marking by m_0^R.

The real system is represented by a model (never created) that fully represents the system under consideration. We observe this real system through observable C-sequences. Ideally, the EXPECTED SYSTEM (our model) completely captures the behavior of the REAL SYSTEM. We will use the notion of real and expected system to define a diagnosis.

Definition 4.1 A subsystem is said to have a FAULT if the language of the real component (G_i^R) is not contained within the language of the corresponding model (G_i^E) of the expected behavior, i.e. $L(G_i^R, m_{0,i}^R) \not\subseteq L(G_i^E, m_{0,i}^E)$ is a fault of component G_i^R.

The following assumption defines consistency requirements between the models of a system and their real world counterparts (items 1 and 2), and it requires that only one fault can occur at a time (item 3).

Assumption 4.1 Model-Reality Relationship Assumption (MRR Assumption):

The real and expected systems are made up of component models such that $\mathcal{G}^R = \{G_1^R, \ldots, G_{n_R}^R\}$ and $\mathcal{G}^E = \{G_1^E, \ldots, G_{n_E}^R\}$ respectively for some positive integers n_R and n_E. For $\mathcal{G}^R$ and $\mathcal{G}^E$ we require:

1. ALL COMPONENTS FOUND IN THE EXPECTED MODEL ALSO EXIST IN THE PHYSICAL SYSTEM WITH THE SAME OUTPUTS: $n_E \leq n_R$, and $C_{out}(G_i^R) = C_{out}(G_i^E)$ for each $1 \leq i \leq n_E$;

2. THE OBSERVED OUTPUTS OF THE REAL SYSTEM AND THE MODEL ARE THE SAME UNDER THEIR INITIAL MARKINGS: Given any $(C_0^R) \in L(\mathcal{G}^R, m_0^R)|_{C_{obs}}$ and $(C_0^E) \in L(\mathcal{G}^E, m_0^E)|_{C_{obs}}$, then $C_0^R \cap C_{out}(\mathcal{G}^R) = C_0^E \cap C_{out}(\mathcal{G}^E)$.

3. THERE EXISTS AT MOST A SINGLE SUBSYSTEM FAULT: there exists at most one i such that $L(G_i^R, m_{0,i}^R) \not\subseteq L(G_i^E, m_{0,i}^E)$.

We specifically note that under our MRR assumption, it is possible for there to be subsystems in the real system, $\mathcal{G}^R$, that have not been modeled in the expected system. This will allow us to diagnose a system even when the system description, $\mathcal{G}^E$, is incomplete in its modeling of the real system's dynamics, $\mathcal{G}^R$. Or more precisely there may exist some G_j^R for $j > n_E$ with some $c \in C_{out}(G_j^R)$ such that $c \in C_{in}(G_i^R) - C_{in}(G_i^E)$ for some $1 \leq i \leq n_E$. In this situation, a diagnosis would return the influenced subsystem (i.e. G_i^R). These unmodeled subsystems may influence subsystems with corresponding models of expected behavior through some unmodeled interconnection. For example, consider figure 4.1 where there exists some unmodeled subsystem, G_5^R, in the real system that is not represented in the model of the system, $\mathcal{G}^E$.

MRR assumption item 3 is the major limitation in this work. In practical systems, a failure of one component or subsystem often leads to other component failures. This issue is complicated by the fact that failure relationships may exist between the subsystems *that are not captured in our condition system model*. As an example consider a failed actuator which then leads to a broken spray head on another subsystem. Clearly these models may not share input or output conditions and hence would be considered causally independent with regards to the condi-

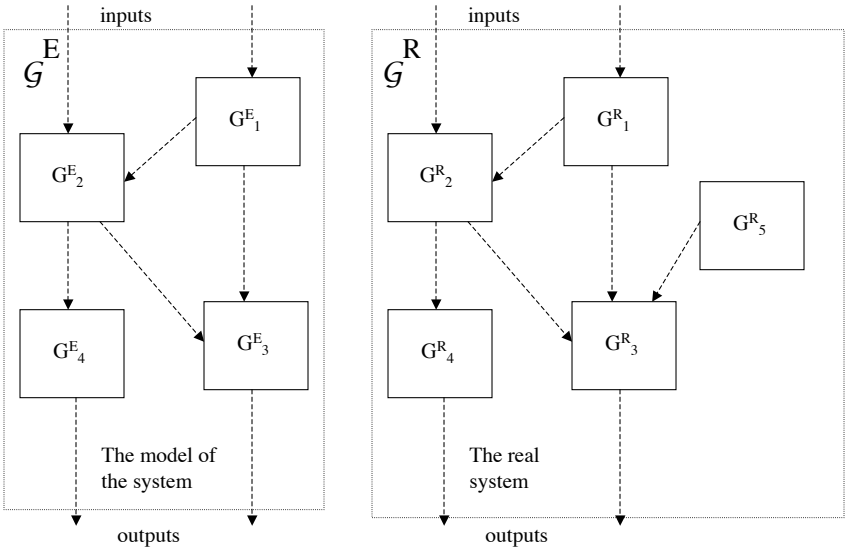

Figure 4.1: A possible situation that occurs when a real component (or interaction) in $\mathcal{G}^R$ is not modeled in $\mathcal{G}^E$.

tion system. Resolution of this issue is important, but it obviously complicates the problem considerably. We also envision it will make the solution much less graceful and intuitive in that it would require further human intervention in terms of specifying these types of physical causal interactions. Our approach requires no extra specification and thus no extra work for the specifier.

Let s_{obs} be an observed C-sequence. A FAULT HAS BEEN DETECTED if it is determined that $s_{obs} \notin L(\mathcal{G}^E, m_0^E)|_{C_{obs}}$, i.e. the observed behavior is not consistent with the expected behavior defined by the model. A FAULT DIAGNOSIS is a localization of where the real system and the model of expected good behavior are inconsistent. This is formally expressed as follows.

Definition 4.2 Consider an observed C-sequence $s_{obs} = (C_0^R \ldots C_{k-1}^R C_k^R)$ such that prior to the last condition set observation, the sequence was consistent with the expected behavior, $(C_0^R \ldots C_{k-1}^R) \in L(\mathcal{G}^E, m_0^E)|_{C_{obs}}$. Define $Diagnosis(s_{obs}) \subseteq \mathcal{G}^E$ such that $G_i^E \in Diagnosis(s_{obs})$ if:

1. The complete observation sequence is not consistent with expected behavior:

$s_{obs} \notin L(\mathcal{G}^E, m_0^E)\,|_{C_{obs}}$, and

2. The observed sequence is consistent with the behavior of relaxing the behavior of subsystem G_i^E :
$s_{obs} \in (L(\mathcal{G}^E/G_i^E, m_0^E))\,|_{C_{obs}}$.

Note from the above definition that we restrict our interest to observed sequences which represent acceptable behavior prior to the most recent observed condition set, when the sequence became no longer representative of correct behavior. Thus, using the terminology of the preceding section, from the above definition, s_{obs} is not within the expected behavior of the plant, but it is in the behavior resulting from relaxing after the initial time the behavior constraints imposed by some subsystem $G_i^E \in Diagnosis(s_{obs})$. In other words, if we allow the subsystem outputs $C_{out}(G_i^E)$ to take on any possible sequences of values (regardless of the subsystem inputs $C_{in}(G_i^E)$ or the dynamics imposed by the model G_i^E) following the initial output, some such sequence of values of $C_{out}(G_i^E)$ would account for the behavior observed (input and output) from the remainder of the subsystems.

We note from the definition above that determining the $Diagnosis(s_{obs})$ set is equivalent to $n + 1$ language inclusion tests for s_{obs}, where n is the number of component models (one inclusion test for statement 1 and n inclusion tests for statement 2). This inclusion testing also yields a detection of a fault occurrence. Given a finite length s_{obs}, each inclusion test can be computed in a finite number of calculations. This relies on the finiteness of the marking space. However, the language inclusion test involves determining paths in the marking reachability graph of the system, and can be quite involved. For this reason, we show how to detect faults utilizing our controller models in chapter 6, and in chapter 7 we introduce a method to diagnose faults by approximating $Diagnosis(s_{obs})$. This approximation method relies on the causal structure of the system, and is suitable for rapid diagnosis of faults.

Also note from the MRR Assumption we assume that at most one subsystem can fail at any time. Thus, our definition of $Diagnosis(s_{obs})$ considers the relaxation of at most one subsystem model.

In [AH02a] and [AH02b], we partition the diagnosis problem into the problem of detection (determining when a fault occurs), and diagnosis (determining the source of the fault). In those papers, determining the source of a fault was the focus. In standard D.E.S. literature, a diagnosis usually refers to both of these activities. Evaluation of definition 4.2 detects and diagnoses faulty behaviors. Evaluation of definition 4.2 item 1 yields a detection of a fault in that a fault occurs if $Diagnosis(s_{obs})$ is non-empty, and is fault free if it is empty. Evaluation of definition 4.2 item 2 determines the set of potentially faulty subsystems.

Example 4.1 Consider the plant in Figure 3.1 and the following observed sequence,[1]

$$s_1 \;=\; (\{AtLeft, GoOff\}, \{AtLeft, GoOff\}\{AtLeft, AtHome, GoOff\})$$

To determine $Diagnosis(s_1)$ directly from its definition, we must consider each subsystem and ask if relaxing its model could give us the observed behavior. We have $Diagnosis(s_1) = \{G_2, G_4\}$, for the following reasons:

G_1 : $G_1 \notin Diagnosis(s_1)$ since under the single fault assumption (MRR Assumption item 1), even if the motor G_1 failed, there is no way that the rest of the system could output $AtLeft$ and $AtHome$ simultaneously.

G_2 : $G_2 \in Diagnosis(s_1)$ since if its model behavior were relaxed, it could potentially output both $Left$ and $Home$, driving G_3 and G_4 to output $AtLeft$ and $AtHome$.

G_3 : $G_3 \notin Diagnosis(s_1)$, since even if G_3 were failed, under the single fault assumption, G_2 should still not get to $Home$ under condition $GoOff$.

[1]Note that the first and second condition sets in the sequence s_1 are equal, so by lemma 3.1, this is equivalent to the sequence consisting of just the first and third sets. However, in this example we keep the second set in to emphasize that although the relaxed behaviors considered in diagnosis must have the initial output consistent with the expected behavior's initial output, the initial output within the sequence does not have to change immediately.

G_4 : $G_4 \in Diagnosis(s_1)$ since G_4 could output $AtHome$ unexpectedly.

Next, consider the sequence

$$s_2 = (\{AtLeft, GoRight\}, \{AtLeft, GoRight\}, \{AtLeft, GoRight, AtHome, AtRight\})$$

We determine that $Diagnosis(s_2) = \{G_2\}$, since under the single fault assumption, only the fault of G_2 could account for $AtLeft$, $AtHome$, and $AtRight$ to be simultaneously true.

Copyright © Jeffrey Ashley 2004

Chapter 5

An Exact Solution to the Diagnosis Problem.

In this chapter, we present a method to determine $\mathtt{Diagnosis}(s_{obs})$ directly by working in a system's marking space. We then show that this method can be evaluated in a finite number of steps. As we shall see though, this process is computationally involved, but it complements a faster (but inexact) method to diagnose a system presented in chapter 7. We envision that a system may employ both methods (when appropriate) in the identification of the source of an observed faulty behavior. These methods also vary in that direct evaluation of the $\mathtt{Diagnosis}(s_{obs})$ as in this chapter also yields a detection of a fault whereas the chapter 7 method does not.

This chapter is organized as follows. In the next section, we present our method for direct evaluation of $\mathtt{Diagnosis}(s_{obs})$. We then show that this method is EFFECTIVELY COMPUTABLE (i.e.- a set of candidate subsystems is returned for a system diagnosis in a finite amount of time).

5.1 An algorithm to determine $\mathtt{Diagnosis}(s_{obs})$ directly

In this section, we present an algorithmic method to determine $\mathtt{Diagnosis}(s_{obs})$. Our main objective is not to present the most efficient method but instead to present a method that we can show is effectively computable. A definition or algorithm is EFFECTIVELY COMPUTABLE if a solution (in our case - a set of candidate subsystems)

can be determined in a finite number of calculations. This is directly analogous to the definition of decidability for Turing machines programs that provide a "yes" or "no" answer.

The following definition is used throughout the remainder of this chapter. It defines a condition set that is CONSISTENT with a marking given some model.

Definition 5.1 For some condition set C and some marking m define the following. C is CONSISTENT with m under C_{obs} if $g_{\mathcal{G}}(m)|_{C_{obs}} = (C \cap C_{out})|_{C_{obs}}$.

In words, a condition set is consistent with a marking when the marking could account for the observed output of the system under consideration (i.e. the observed output is the condition set C' intersected with the set of observable outputs). This is a realistic assumption that requires that a model's observed outputs be generated when a model is considered for some marking.

First, we define three modified reachability definitions. The first is the set of markings immediately reachable given some observable condition set, C. The last two define sets of possible markings such that the set of observed output conditions do not change.

Definition 5.2 Define the OBSERVABLE IMMEDIATELY REACHABLE MARKING SET $R^1(m, C)$ for some marking m and the observable condition set $C \subseteq C_{obs}$ as the set of all immediately reachable markings :

$R^1(m, C) := \{m' \in f_{\mathcal{G}}(m, C') \mid \text{ for some } C' \text{ such that } C'|_{C_{obs}} = C\}$

Definition 5.2 defines the set of next state markings immediately reachable given the constraint imposed by the observable condition set, C.

In contrast to Definition 5.2 of $R^1(m, C)$ which defines the set of ALL markings immediately reachable given C, $R^1_{const}(m, C)$ given below defines the set of immediately reachable markings from m such that the observable conditions of the system DO NOT CHANGE. We note that within this set of markings, some conditions of the system may be changing, but these conditions would be unobservable.

Definition 5.3 Define the OBSERVABLY INVARIANT IMMEDIATELY REACHABLE MARK-

ING SET $R^1_{const}(m, C)$ for some marking m and the observable condition set $C \subseteq C_{obs}$ as:

$$R^1_{const}(m, C) := \{m' \in f_{\mathcal{G}}(m, C') \mid C'|_{C_{obs}} = C \text{ and } m' \text{ is consistent with } C\}$$

By repetitively applying the previous definition, we extend this idea to complete reachability for the observably invariant marking set.

Definition 5.4 Given some $C \subseteq C_{obs}$, $R^\infty_{const}(m, C)$ is the set of markings such that:

1. $m \in R^\infty_{const}(m, C)$ and
2. for each $m' \in R^\infty_{const}(m, C)$, then $R^1_{const}(m', C) \subseteq R^\infty_{const}(m, C)$

We note that $R^\infty_{const}(m, C)$ does not necessarily equal the set of markings that could correspond to the given set C. This is because there may be some markings that yield an observation consistent with C that are only reachable from a marking sequence that does not output C for every marking in the sequence.

To simplify the following, we assume that the C-sequences we consider are TRIM. A trim C-sequence is of minimal length for its respective equivalence class. For a trim C-Sequence, $(C_1, C_2, \ldots, C_k)$ the following holds true, $C_i \neq C_{i+1}$ for $1 \leq i \leq k-1$. Or, each successive condition in a trim C-Sequence contains new information about the system. We note that we can easily make any such C-sequence trim by repetitively removing repetitive condition sets. This follows from lemma 3.1.

We use a trim observable C-sequence, a condition system and an initial marking to repetitively build sets of markings that could correspond to the observations of the system (the condition sets within the C-sequence). We know we have a failure whenever the set of markings corresponding to some observed condition set is empty.

We will use the recursive definition as defined below in the final result of this chapter. In essence, we define sequences of marking sets corresponding to sets of observable conditions given an initial marking. As new condition sets are observed, we will construct these marking sets and use them to determine whether the observable C-sequence remains in the expected language of the system. In this

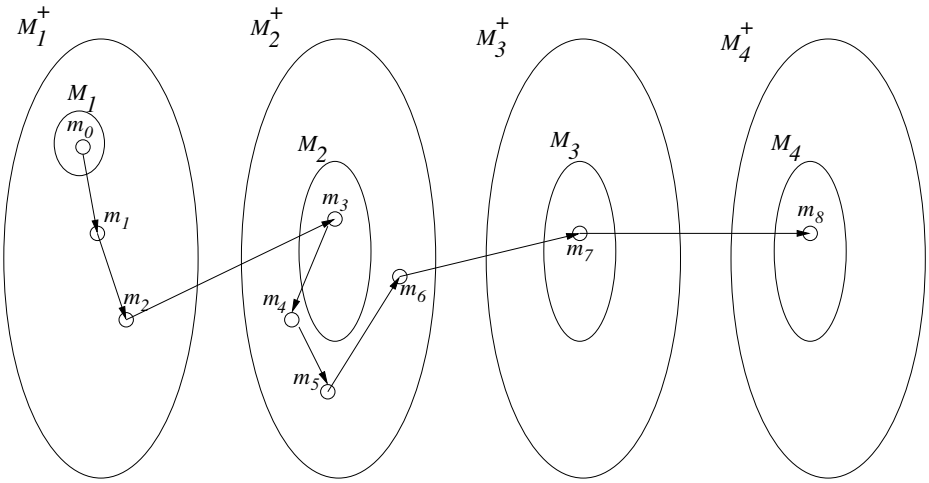

Figure 5.1: An Illustration of How States Evolve in a Marking Space partitioned into the sets M_i and M_i^+.

way we can determine $\mathtt{Diagnosis}(s_{obs})$, hence showing definition 4.2 is effectively computable.

Definition 5.5 Given a condition system $\mathcal{G}$, an initial marking m_0, and a trim observable C-sequence $(C_1, C_2, \ldots, C_n)$ such that C_1 is consistent with m_0, recursively define the sets M_i and M_i^+ as:

1. $M_1 := \{m_0\}$.
2. For $1 \leq i \leq n$, $M_i^+ := \{m' \mid C_i \text{ is consistent with } m' \text{ and } m' \in R^\infty_{const}(m, C_i) \text{ for some } m \in M_i\}$.
3. For $1 \leq i \leq n-1$, $M_{i+1} := \{m' \mid C_{i+1} \text{ is consistent with } m'$ and $\exists m \in M_i^+$ such that $m' \in R^1(m, C_i)\}$.

The set, M_{i+1}, corresponds to a HOP-IN set of markings in that the system could have transitioned from some marking that had an observation of C_i to one of the hop-in set of markings that has an observation of C_{i+1}. This is the set of transition markings that could account for the change in observation.

The set, M_i^+ is an EXPANSION set that contains M_i plus markings reachable from markings in M_i such that the observed conditions do not change. Figure 5.1 shows

how an example trace of markings in the set of all markings would translate to membership in the sets M_i and M_i^+.

Now we are ready to present an algorithm that can be used to determine $Diagnosis(s_{obs})$. Algorithm 5.1 determines whether some observed C-Sequence can be generated by the system given its initial marking. This algorithm can be used directly to determine (in a finite number of steps) item 1 of the $Diagnosis(s_{obs})$ definition. In the next section we show this and we also show how to use it to determine item 2 of this definition given some additional steps and calculations. We also show that these additional steps and calculations are effectively computable.

Algorithm 1: An algorithm to determine if an observed sequence is in the expected language of the system.

Input: A trim $s_{obs} = (C_1^R \dots C_{k-1}^R C_k^R)$, $\mathcal{G}^E$, and m_0^E.

Output: A YES or NO answer to the assertion $s_{obs} \in L(\mathcal{G}^E, m_0^E)\,|_{C_{obs}}$

(1) $index = 1$
(2) $M_1 = \{m_0^E\}$
(3) **while** $(index < k)\{$
(4) Given C_{index}^R calculate M_{index}^+
(5) Given C_{index}^R and $C_{index+1}^R$, calculate $M_{index+1}$
(6) **if** $(M_{index+1} = \emptyset)$
(7) **return** NO.
(8) $index = index + 1$
(9) $\}$
(10) **return** YES.

Theorem 5.1 Given a trim $s_{obs} = (C_1^R \dots C_{k-1}^R C_k^R)$, $\mathcal{G}^E$, and m_0^E then algorithm 5.1 returns a "YES" if and only if $s_{obs} \in L(\mathcal{G}^E, m_0^E)\,|_{C_{obs}}$.

Proof: First note that by the MRR Assumption for $index = 1$ then $s_{obs} \in L(\mathcal{G}^E, m_0^E)\,|_{C_{obs}}$ by definition. Also note that for $k = 1$, then by the MRR Assumption algorithm 5.1 returns "YES" by definition.

(Only If) It suffices to show that if $index = k$ and $s_{obs} \notin L(\mathcal{G}^E, m_0^E)\,|_{C_{obs}}$, then algorithm 5.1 returns "NO".

Let $s_{obs} = (C_1^R \ldots C_{k-1}^R)$. Let's assume for $index = k-1$ that $s'_{obs} \in L(\mathcal{G}^E, m_0^E)\,|_{C_{obs}}$, and for $index = k$ that $s_{obs} \notin L(\mathcal{G}^E, m_0^E)\,|_{C_{obs}}$. For s'_{obs} there must exist some marking sequence that is consistent with s_{obs} for every marking in the sequence and so by definition 5.5 there will be a non-empty $M_{index+1}$ for each $0 \leq index \leq k-2$, and the algorithm will not have exited. Now for $index = k$, since $s_{obs} \notin L(\mathcal{G}^E, m_0^E)\,|_{C_{obs}}$ then there does not exist a next state marking from any of the possible markings from M_{k-1}^+ that outputs C_k^R and hence $M_{index+1}$ for $index = k-1$ would be empty, which would make the algorithm exit with a "NO".

(If) Let's assume for $index = k-1$ that $s'_{obs} \in L(\mathcal{G}^E, m_0^E)\,|_{C_{obs}}$, and for $index = k$ that $s_{obs} \in L(\mathcal{G}^E, m_0^E)\,|_{C_{obs}}$. For s'_{obs} there must exist some marking sequence that is consistent with s_{obs} for every marking in the sequence and so by definition 5.5 there will be a non-empty $M_{index+1}$ for each $0 \leq index \leq k-2$, and the algorithm will not have exited. Now, for $index = k$, if $s_{obs} \in L(\mathcal{G}^E, m_0^E)\,|_{C_{obs}}$ then there must exist a next state marking from at least one of the possible markings from M_{k-1}^+ that outputs C_k^R and hence $M_{index+1}$ for $index = k-1$ would be non-empty. Since $index = k$ the algorithm will exit with a "YES". □□□

Our intention is to use this algorithm for evaluation of definition 4.2 item one directly. This part of the $Diagnosis(s_{obs})$ definition yields a detection of an observed fault.

We also can use this algorithm for evaluation definition 4.2 item two given some extra work. This portion of the $Diagnosis(s_{obs})$ definition is concerned with the diagnosis of the fault(i.e. localization of faulty sub-system). To implement definition 4.2 item two using this algorithm we need to relax each sub-system, in turn, and use algorithm 5.1 on the new system model. This will need to be done for each sub-system individually, so if there are 10 sub-systems in the system model, then we would apply this algorithm 10 times, once for each new system model where one of the sub-systems relaxed.

In the next section, we show that the algorithm presented yields an evaluation

of $\mathtt{Diagnosis}(s_{obs})$ and that this can be computed in a finite number of steps.

5.2 $\mathtt{Diagnosis}(s_{obs})$ is effectively computable

In this section we show that definition 4.2 can be directly evaluated for a system in a finite number of calculations. As we shall see though, an exact solution is computationally complex in general, although it provides a best possible diagnosis within the constraints of observability.

First note that finding if C is consistent with m is a simple set intersection on finite sets and is hence effectively computable. Also, finding the sets $R^1(m, C)$ and $R^1_{const}(m, C)$ is effectively computable. Given some marking, m, we can directly determine $R^1(m, C)$ and $R^1_{const}(m, C)$ by considering each marking in $M_{\mathcal{G}}$ to determine if it meets the requirements of definition 5.3.

The following lemma shows that $R^\infty_{const}(m, C)$ is effectively computable.

Lemma 5.1 Given some condition system, a marking, m and a corresponding observable condition set, $C \subseteq C_{obs}$ satisfying the MRR Assumption, $R^\infty_{const}(m, C)$ is effectively computable.

Proof:

We show that applying this definition is effectively computable by sketching out an algorithm to determine $R^\infty_{const}(m, C)$. Given the starting marking, $m \in M_{\mathcal{G}}$, define set R with $R = \{m\}$. Determine $R^1(m, C)$ which requires at most $|M_{\mathcal{G}}| - 1$ states to consider and the add these markings to R. Then R has been determined in a finite number of steps. If there is some new state $m' \in R$, we then need to find $R^1(m', C)$ which requires at most $|M_{\mathcal{G}}| - 2$ states to consider for inclusion in R .

We can repetitively apply this logic until we have either added all states in $M_{\mathcal{G}}$ to R, at which point we can stop, or if we have checked all states in R and have no other states to consider then we can stop. The resulting R is $R^\infty_{const}(m, C)$. In either case, since this is a summation series, it would take at most $(|M_{\mathcal{G}}| - 1) \times (|M_{\mathcal{G}}|)/2$ comparisons to determine $R^\infty_{const}(m, C)$ and is hence effectively computable. □□□

We note that the marking space is typically very large and so determination of reachability would be computationally involved. We believe though that the structure we have imposed on our models may lend itself to potential reductions in the complexity. For example, any state that is not consistent with C can immediately be excluded from consideration. We can also exploit other structural requirements of our models to further improve the complexity.

Lemma 5.2 Given a condition system $\mathcal{G}$, a marking m_0, and a C-sequence $(C_1, C_2, \ldots, C_n)$ for some $n < \infty$ as defined in definition 5.5, determining the sets M_i and M_i^+ for $1 \leq i \leq n$ is effectively computable.

Proof:

We need to construct $2n$ sets corresponding to M_i and M_i^+ for all i. M_1 is given, and M_1^+ is effectively computable since we consider a finite number of m' and each step in determining membership in the set is effectively computable. In a similar manner, finding M_2 from M_1^+ is effectively computable. Following this logic we can incrementally build the M_i and M_i^+ for all $i \leq n$ in a finite number of steps. Hence the lemma is proved. □□□

The proof of the computability of definition 4.2, will use the sets M_k and M_k^+ to determine whether the next condition set, C_k, in some observable and trim C-sequence places the C-sequence out of the expected language for the system. If the C-sequence is not in the expected language of the system, the sets M_k and M_k^+ will also be used to identify the set of subsystems that could account for the deviation from expected behavior.

We will first show that item 1 of definition 4.2 is effectively computable($s_{obs} \notin L(\mathcal{G}^E, m_0^E)\,|_{C_{obs}}$), then extend this proof to item 2.

Lemma 5.3 Consider some condition system $\mathcal{G}^E$, m_0^E and a trim observable C-sequence $s_{obs} = (C_1^R \ldots C_{k-1}^R C_k^R)$, and an initial marking m_0 such that C_1^R is consistent with m_0^E, then definition 4.2 item 1, $s_{obs} \notin L(\mathcal{G}^E, m_0^E)\,|_{C_{obs}}$, is effectively computable for for some finite k.

Proof:

Use induction on the length of the trim observable C-sequence $s_{obs} = (C_1^R \dots C_{k-1}^R C_k^R)$. For the base case, $|s_{obs}| = 1$ and $s_{obs} = (C_1^R)$. We note that by the MRR assumption item 2, that the expected and the real system are initially consistent and so s_{obs} is in the expected language by default. We need to also build, M_1 and M_1^+, which were shown to be effectively computable.

Now assume for some $k > 1$ and s'_{obs} where $|s'_{obs}| = k-1$ and we have incrementally built M_i and M_i^+ for all $i \leq k-1$, $s'_{obs} \in L(\mathcal{G}^E, m_0^E)|_{C_{obs}}$ is effectively computable. We need to show that $s_{obs} = (C_1^R \dots C_{k-1}^R C_k^R) \in L(\mathcal{G}^E, m_0^E)|_{C_{obs}}$ is effectively computable.

For the C-sequence such that $|s'_{obs}| = k-1$, the test, $s'_{obs} \notin L(\mathcal{G}^E, m_0^E)\ |_{C_{obs}}$, yields one of two results. If, $s'_{obs} \notin L(\mathcal{G}^E, m_0^E)\ |_{C_{obs}}$ for $|s'_{obs}| = k-1$, then we know for $|s_{obs}| = k$, $s_{obs} \notin L(\mathcal{G}^E, m_0^E)\ |_{C_{obs}}$ as well, and hence it is effectively computable. If, $s'_{obs} \in L(\mathcal{G}^E, m_0^E)\ |_{C_{obs}}$ for $|s_{obs}| = k-1$, then to determine if s_{obs} is in the expected language we need to find, M_k(which was shown effectively computable). $s_{obs} \notin L(\mathcal{G}^E, m_0^E)\ |_{C_{obs}}$ if M_k is an empty set. Our logic is as follows: if M_k is empty then there are no valid next states that allow the next observable condition set to be C_k^R and hence the observation cannot be in the expected language. Since determination of M_k is effectively computable, $s_{obs} \notin L(\mathcal{G}^E, m_0^E)\ |_{C_{obs}}$ is effectively computable. □□□

Lemma 5.4 Consider $\mathcal{G}^E$, m_0^E and a trim observable C-sequence $s_{obs} = (C_1^R \dots C_{k-1}^R C_k^R)$ for some finite k such that C_1^R is consistent with m_0^E, then definition 4.2 item 2, $s_{obs} \in (L(\mathcal{G}^E/G_i^E, m_0^E))\ |_{C_{obs}}$ is effectively computable given some condition system $\mathcal{G}^E$ relaxed with respect to subsystem G_i.

Proof:

We will show $s_{obs} \in (L(\mathcal{G}^E/G_i^E, m_0^E))\ |_{C_{obs}}$ is effectively computable by constructing a system $\mathcal{G}'$ that generates the same language as the system $\mathcal{G}^E$ that is relaxed with respect to G_i. We will generate $\mathcal{G}'$ by replacing G_i^E with $G_{i,NEW}^E$ with initial marking $m_{0,i,NEW}^E$ that implements the relaxed behavior. The other subsystem mod-

els will remain unchanged and retain their initial markings.

Clearly the new system $\mathcal{G}'$ generates the same language as the system $\mathcal{G}$ relaxed with respect to G_i^E in that $G_{i,NEW}^E$ can generate any output sequence regardless of the input conditions. By definition then, the new system $\mathcal{G}'$ under its new initial marking can generate the relaxed language of the system.

So to determine if $s_{obs} \in (L(\mathcal{G}^E/G_i^E, m_0^E))\mid_{C_{obs}}$ we generate a new system $\mathcal{G}'$ and apply the result of lemma 5.3 to see if s_{obs} is in the language of the new system. Since both steps are effectively computable, the lemma is proved. □□□

In theorem 5.2, we show that the $Diagnosis(s_{obs})$ can be computed in a finite number of steps.

Theorem 5.2 Given $\mathcal{G}^E$, m_0^E and a trim observable C-sequence $s_{obs} = (C_1^R \dots C_{k-1}^R C_k^R)$ for some finite k, such that C_1^R is consistent with m_0, definition 4.2 is effectively computable.

Proof: In lemma 5.3 we showed that $s_{obs} \notin L(\mathcal{G}^E, m_0^E)\mid_{C_{obs}}$ is effectively computable. In lemma 5.4 we showed that $s_{obs} \in (L(\mathcal{G}^E/G_i^E, m_0^E))\mid_{C_{obs}}$ for any G_i^E in $\mathcal{G}$ is effectively computable. We note that there are a finite number of G_i^E in $\mathcal{G}$, and to determine $Diagnosis(s_{obs})$ we would need to directly evaluate item 1 once, and item 2 once for each subsystem in $\mathcal{G}$. Hence the theorem is proved. □□□

We do not necessarily need to find $M_{\mathcal{G}}$ in its entirety to determine a diagnosis. In future research, we will develop techniques to exploit the structure to improve the computational cost of the method presented. We envision that we would build the sets, M_i and M_i^+, as new observations of the system are encountered.

In this chapter, we showed an algorithm to determine the marking sets consistent with some observed C-Sequence. This is done by considering marking sets that are consistent with each individual condition set from the C-Sequence given the initial marking. This method can be used for fault detection by identifying when some marking set, for a given condition set from the C-Sequence, is empty. It can also then be used for a fault diagnosis by relaxing each subsystem in the system and

reapplying the fault detection method to see if this subsystem belongs to the diagnosis set. The diagnosis set is the set of subsystems, when relaxed, yield non-empty marking sets for each condition set in the C-Sequence. This method was also shown to be effectively computable.

Copyright © Jeffrey Ashley 2004

Chapter 6

A Fault Detection Scheme

In [HGSA00] we presented a method to synthesize a controller model called a taskblock to control some subsystem model. These models can then be used to generate control code necessary to drive some subsystem to a desired target state. We also showed under what situations these taskblocks could be combined to drive an entire system to some target state. In this chapter, we introduce a method to add a detector of unexpected behaviors into this framework. We show that this method detects faults of the subsystem model.

A taskblock is a form of a condition system that can be generated (under certain assumptions) given a model of a subsystem and a specification of desired behavior from this subsystem. For this chapter we approach the detection problem (i.e.- determining a fault has occurred) by appending unexpected responses from the subsystem (under direction from a taskblock) into the taskblock itself. When an unexpected response is detected, the taskblock then moves to a fault state.

One objective of this chapter is to generalize the discussion about taskblock synthesis in an effort to simplify the discussion and to make presentation of the key idea from this chapter clearer. One method of taskblock generation is presented in [HGSA00]. The other objective is to introduce a method to transform a taskblock guaranteed to work under certain conditions into a taskblock that also detects faulty behaviors.

The remainder of the chapter is organized as follows. First we will review taskblocks. This chapter varies from the section from [HGSA00] in that we will include the notions of the "expected" and the "real" system. Next we present the a generalized description of taskblock synthesis and the limitations we assume for

this chapter. We then present the fault detection scheme, and show that the desired behavior of the taskblock is preserved (this is tied to the notion of an EFFECTIVE taskblock) and that we do in fact detect faulty behaviors. We conclude this chapter with a discussion that includes some areas of future research.

6.1 A block diagram perspective of taskblocks.

In [HGSA00] we were interested in using subsystem models(in this chapter also a component model) to develop controllers that would drive a system to a targeted state. In this work, these controllers are generated by a set of connected taskblocks that are generated by analysis of the subsystem models. We did not consider faulty behavior and so the basic assumption was that the real system would behave exactly as expected(i.e. $L(G_i^E, m_0^E) = L(G_i^R, m_0^R)$ using the notation of the previous chapters). This section has been modified from [HGSA00] in that we have introduced the ideas of "expected" and "real" systems.

The plants that we consider to be controlled are modeled by collections of condition models representing the components of the plant. Let this set of condition models representing components be denoted as $\mathcal{G}_{compo}^E$. These represent the subsystem models of the plant and are used in controller synthesis. Let the set real system components (represented by $\mathcal{G}_{compo}^E$) be denoted by $\mathcal{G}_{compo}^R$. As in previous chapters, this is an idealized and never implemented abstraction of the actual system.

The controllers that we consider are also represented as collections of condition models. The set of these controller models, representing elements of the control logic, are called TASKBLOCKS, and are denoted as the set $\mathcal{G}_{tasks}$. A system $\mathcal{G}$ then can consist of a collection of both component models and taskblocks operating together. For control synthesis define the system as $\mathcal{G}^E \subseteq \mathcal{G}_{compo}^E \cup \mathcal{G}_{tasks}$, and define the system while in actual use (i.e. during control of the real system) as $\mathcal{G}^R \subseteq \mathcal{G}_{compo}^R \cup \mathcal{G}_{tasks}$.

Please note that in the other chapters of this dissertation we used the term SUBSYSTEM to represent a single condition system and the set of all of these subsystems was denoted by $\mathcal{G}$ (in other work in this dissertation we have only considered open

loop models and this notation was succinct). In this chapter, a COMPONENT model is the subsystem model of previous chapters. We have chosen to stay with the notation from [HGSA00] in order to make this more accessible to people familiar with that paper.

Each taskblock has a specific control function. Let x denote the intended control function. In [HGSA00], such a control function may represent either driving the system to a given condition or controlling the system to maintain a condition as true. The input and output structure of a taskblock is shown in figure 6.1. A taskblock becomes ACTIVATED to begin its control function upon its ACTIVATION CONDITION, which uniquely identifies the taskblock. Let $C_{do} \subset \mathit{AllC}$ be the set of ACTIVATION CONDITIONS associated with taskblocks. For each element $do_x \in C_{do}$ we associate the following:

- $TB(do_x) \in \mathcal{G}_{tasks}$ is the unique taskblock (condition system model) for which $do_x \in C_{in}(TB(do_x))$. No other taskblocks or components have do_x as an input.
- $compl(do_x) \in C_{out}(TB(do_x))$ is a condition output from the taskblock, indicating task completion.
- $idle(do_x) \in C_{out}(TB(do_x))$ is a condition output from the taskblock and indicates that the taskblock is not activated. There exists exactly one place p in $TB(do_x)$ for which $idle(do_x)$ is an output, and furthermore, it is the only output of that place, $\Phi_{TB(do_x)}(p) = \{idle(do_x)\}$. In all subsequent discussion, we will assume each task block has only this place marked under any initial marking considered.
- $G_{compo}(do_x) \in \mathcal{G}^E_{compo}$ is a component model associated with the task do_x. The same component model may be associated with many different tasks. To simplify later discussion, when the activation condition has a subscript indicating its goal (such as do_x for $goal(do_x) = x$), and a unique component outputs that condition, then we use the subscript to indicate the component net which outputs the target. Thus, $G_{compo}(do_x) = G^E_x$ is the net which outputs condition x.

- $goal(do_x) \in C_{out}(G_{compo}(do_x))$ is a condition output from the component model.
- $C_{init}(do_x) \subseteq C_{in}(TB(do_x)) \cap C_{out}(G_{compo}(do_x))$ is a set of INITIATION CONDITIONS for the taskblock that are output from the component $G_{compo}(do_x)$.

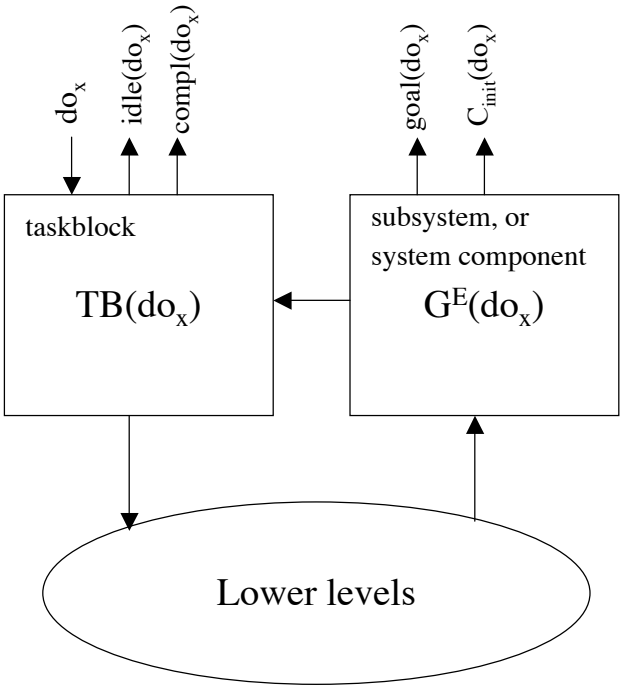

Figure 6.1: The relationship between a taskblock, its associated system component, and the associated conditions.

Activation conditions are only associated with taskblocks. These conditions come from a higher level supervisor and do not communicate with component models directly. We note however, that in our hierarchical scheme, the higher level supervisor can be another taskblock. We will call the conditions $idle(do_x)$ and $compl(do_x)$ STATUS CONDITIONS. They are used for taskblock to supervisor communication, and they do not communicate to the open loop system directly(either the real or expected system). Also note that the $goal(do_x)$ and $C_{init}(do_x)$ (the output of $G_{compo}(do_x)$ from the figure) are merely output conditions of the component

G^E_{compo} and do not represent new conditions appended to the model.

We interpret a taskblock as follows: The output condition $idle(do_x)$ indicates that the taskblock is not currently outputting any other conditions. A taskblock $TB(do_x)$ becomes ACTIVE (and thus $idle(do_x)$ becomes false) upon the conditions $\{do_x\} \cup C_{init}(do_x)$ becoming all true. As long as do_x remains true, the taskblock and system component will interact until eventually the condition $goal(do_x)$ is output from the component model $G_{compo}(do_x)$ and the condition $compl(do_x)$ is output from the task block, indicating completion of the task. Whenever do_x becomes false, the taskblock returns to the idle state. The following definition of EFFECTIVE formally describes the behavior of a taskblock when it is interacting with a system in its intended manner.

Definition 6.1 Given a system $\mathcal{G}^E \subseteq \mathcal{G}_{tasks} \cup \mathcal{G}^E_{compo}$ with initial state m^E_0 and a condition $do_x \in C_{in}(\mathcal{G}^E) \cap C_{do_x}$ such that $idle(do_x) \in g(m^E_0)$, do_x is EFFECTIVE FOR CONTROL for $\mathcal{G}^E$ under m^E_0 if each of the following statements are true:

1. CONTINUED ACTIVATION IMPLIES EVENTUAL COMPLETION: For all $s \in L(\mathcal{G}^E, m^E_0)$, if $(\emptyset\ (\{do_x\} \cup C_{init}(do_x))) \leq s$, then for any set $C_{ext} \subseteq AllC$ such that $do_x \in C_{ext}$ and $C_{ext} \cap (C_{out}(\mathcal{G}^E) \cup \{\neg do_x\}) = \emptyset$, there exists s' such that $ss' \in L(\mathcal{G}^E, m^E_0)$, $(C_{ext}) \leq s'$, and

   $$(\{do_x\}\,\{do_x, compl(do_x)\}) \leq s'$$

2. COMPLETION IMPLIES EARLIER ACTIVATION: For all $s \in L(\mathcal{G}^E, m^E_0)$, if $(\emptyset\{compl(do_x)\}) \leq s$, then

   $$(\emptyset\ (\{do_x\} \cup C_{init}(do_x)\ )\ \emptyset) \leq s$$

3. COMPLETION IMPLIES ACHIEVED GOAL: For any condition set string s and any condition set C such that $sC \in L(\mathcal{G}^E, m^E_0)$, if $\{compl(do_x)\} \subset C$, then

   $$\{compl(do_x), goal(do_x)\} \subseteq C$$

4. LEAVING COMPLETION IMPLIES EARLIER DEACTIVATION: For all $s \in L(\mathcal{G}^E, m^E_0)$, if $(\emptyset\{compl(do_x)\}\{\neg compl(do_x)\}) \leq s$, then

   $$(\emptyset\,\{\neg do_x\}\,\emptyset) \leq s$$

5. DEACTIVATION IMPLIES EVENTUAL RETURN TO IDLE: For all $s \in L(\mathcal{G}^E, m_0^E)$, if $(\emptyset\{\neg do_x\}) \leq s$, for any set $C_{ext} \subseteq \mathit{AllC}$ such that $\neg do_x \in C_{ext}$ and $C_{ext} \cap (C_{out}(\mathcal{G}^E) \cup \{do_x\}) = \emptyset$, there exists s' such that $ss' \in L(\mathcal{G}^E, m_0^E)$, $(C_{ext}) \leq s'$, and

$$(\{\neg do_x\}\{\neg do_x, idle(do_x)\}) \leq s'$$

The first statement states that after do_x and $C_{init}(do_x)$ conditions are true, if do_x remains true, then there will eventually follow a completion condition $compl(do_x)$ from the task block. Since the statement must be true for any C_{ext} such that $do_x \in C_{ext}$ and $C_{ext} \cap (C_{out}(\mathcal{G}^E) \cup \{\neg do_x\}) = \emptyset$, then after the initial $C_{init}(do_x)$, no external signal (other than do_x) from beyond the taskblock or the component model is required to reach completion. Therefore, completion is reached entirely through the interaction of the taskblocks and components in $\mathcal{G}^E$, and not from any other external conditions.

The second statement of the definition states that if $compl(do_x)$ is true at the end of s, then at some prior time in s the conditions do_x and $C_{init}(do_x)$ were true. The third statement in the definition states that whenever do_x and $compl(do_x)$ are simultaneously true, then $goal(do_x)$, output from component $G_{compo}(do_x)$, must be true also. Statement 4 says that once a taskblock achieves completion, it will stay there until do_x becomes false. Finally, the last statement says that if do_x is false, then eventually $idle(do_x)$ will become true.

The definition above can be expanded in the obvious manner to sets of conditions $C' \subseteq C_{in}(\mathcal{G}^E) \cap C_{do_x}$ by replacing occurrences of do_x with all elements of C', occurrences of $compl(do_x)$ with all the set of completion conditions corresponding to elements of C', and occurrences of $goal(do_x)$ with the set of goal conditions corresponding to elements of C'. Thus, for example, for $C' = \{do_x, do_y\}$, statement 1 would imply that following the simultaneous activation of both $TB(do_x)$ and $TB(do_y)$, eventually both $compl(do_x)$ and $compl(do_y)$ must be simultaneously true.

Example 6.1 Figure 6.2 illustrates the ideas presented above. The activation condition do^A_{mid} has associated with it a taskblock $TB(do^A_{mid})$ and a component model

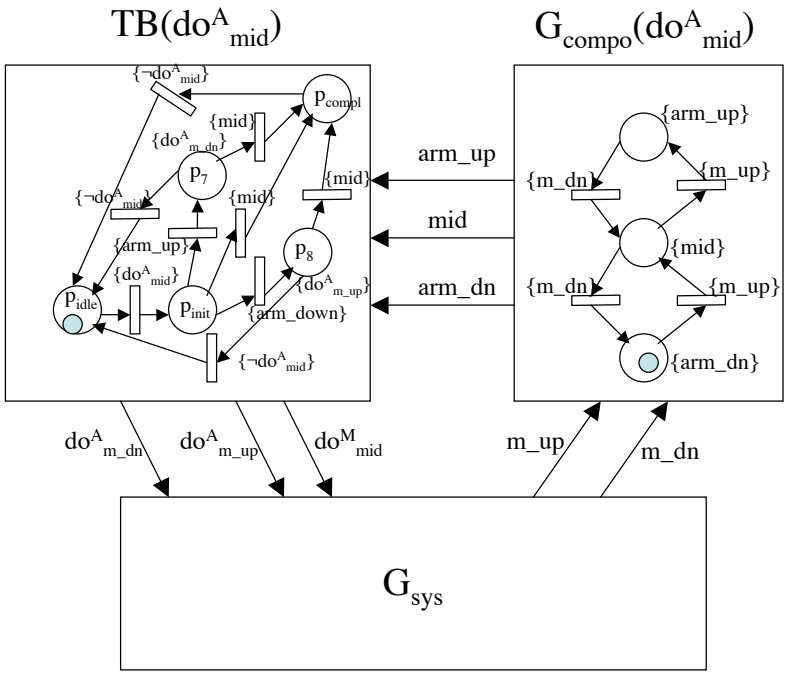

Figure 6.2: An example of effective do_x.

$G_{\mathrm{compo}}(\mathrm{do}^{A}_{\mathrm{mid}})$. (The superscript "A" in the condition $\mathrm{do}^{A}_{\mathrm{mid}}$ will be explained in section 6.2. It is not relevant to the discussion at this point.) The component model represents a robot position as either $\mathrm{arm\_up}$, mid, or $\mathrm{arm\_down}$. To move the arm requires conditions $\mathrm{m\_up}$ or $\mathrm{m\_down}$, indicating motor on up and motor on down. We have $\mathrm{goal}(\mathrm{do}^{A}_{\mathrm{mid}}) = \mathrm{mid}$.

The taskblock has an initial state with p_{idle} marked and condition output $\mathrm{idle}(\mathrm{do}^{A}_{\mathrm{mid}})$. Upon receipt of the activation signal $\mathrm{do}^{A}_{\mathrm{mid}}$, the marking changes and is no longer idle. Since the component has place p_3 marked and outputs condition $\mathrm{arm\_down}$, then the taskblock marking moves from p_{idle} to p_{init} to p_8. From that state, the taskblock outputs condition $\mathrm{do}^{A}_{\mathrm{m\_up}}$. If that condition is effective for G_{sys}, then by definition 6.6, G_{sys} eventually outputs condition $\mathrm{goal}(\mathrm{do}^{A}_{\mathrm{m\_up}}) = \mathrm{m\_up}$. This then enables a transition in the component model, which then changes state and outputs condition mid. This enables a transition in the taskblock, allowing the taskblock to change state to p_{cmpl} which outputs condition $\mathrm{compl}(\mathrm{do}^{A}_{\mathrm{mid}})$, indicat-

ing completion.

We should note that this is a simple example, where only the condition m_up was necessary to move the component to the desired goal state. In a more complex example, the taskblock would have to output sequences of conditions to step the component towards the goal, and there could possibly be multiple places in the taskblock which could output the completion condition.

We can illustrate each of the requirements of EFFECTIVE for this example. Note in the example that the signal do^{A}_{mid} implies eventual completion because of the manner in which the taskblock and its component work together. Similarly, the taskblock only reaches its completion state after it has been activated. Furthermore, reaching the completion state p_{cmpl} can only occur after the component has reached $goal(do^{A}_{mid}) = mid$. Upon completion, the taskblock outputs a condition do^{M}_{mid} (explained in the following section), which activates a different taskblock responsible for maintaining the component outputting condition mid. Thus the component will continue to output the goal as long as the place p_{cmpl} is marked. Finally, we note that when the taskblock is deactivated with signal $\neg do^{A}_{mid}$, it returns to its idle state.

In our analysis of taskblocks, it is important for us to assume that the controller reacts faster than the component that it controls. This leads us to the following definition.

Definition 6.2 Consider a system $\mathcal{G}^{E}$, an external input condition set C_{ext} such that $C_{ext} \cap C_{out}(\mathcal{G}^{E}) = \emptyset$, and a condition $do_x \in C_{do_x}$ such that $TB(do_x), G_{compo}(do_x) \in \mathcal{G}^{E}$. A marking m of $\mathcal{G}^{E}$ is a CONTROL-WAIT STATE for $TB(do_x)$ under C_{ext} if for any transition t in $TB(do_x)$ state-enabled under m, there exists some $c \in C_{\mathcal{G}}(t)$ such that:

1. if $c \in C_{out}(\mathcal{G}^{E})$, then c is not output from $\mathcal{G}^{E}$ under m^{E}, or else,
2. if $c \notin C_{out}(\mathcal{G}^{E})$, then $\neg c \in C_{ext}$.

The implication is that when the system $\mathcal{G}^{E}$ is in a control-wait state, the taskblock must wait for an external signal or for the component $G_{compo}(do_x)$ to

change state before proceeding to a next state of the taskblock. Thus, the taskblock has reached a state in which it cannot proceed farther. This allows us to present the following assumption to be used through the rest of the dissertation.

Prompt Controller Assumption (PCA): For any system $\mathcal{G}^R$ and marking m^R and external input condition set C_{ext}, transitions in components in $\mathcal{G}^E$ fire only if m^R is a control-wait state under C_{ext} for each taskblock in $\mathcal{G}^R$.

Finally, before discussing the interaction of taskblocks, we must introduce the following property that states that a taskblock will not output a signal do_x unless the initiation conditions $C_{init}(do_x)$ are assured to be already true. This property can be assured through the structure of the taskblock, as shown in section 6.2.

Definition 6.3 Given a system $\mathcal{G}^E$, a task block $TB' \in \mathcal{G}^E$ is WELL-STRUCTURED under $\mathcal{G}^E$ if for every activation signal $do_x \in C_{out}(TB')$, the initial conditions $C_{init}(do_x)$ are necessarily true whenever do_x becomes true from TB'.

6.2 A generalized discussion of taskblock models

In this section, we consider the modeling details of taskblocks. In [HGSA00] we present methods for synthesizing taskblocks. Here, we wish to present the ideas of that paper in a generic way without going into the details of synthesis. For each component model and each output condition of the components, we consider two types of taskblocks. The first type is called a MAINTAIN-TYPE, and its purpose is to keep a condition of the system true, given that it was already true when the taskblock was activated. The second type is called an ACTION-TYPE. Its purpose is to drive the system to a given condition from any initial state. For a given condition x, we distinguish between the action-type and maintain-type taskblocks through the activation signals: do_x^A is the activation condition for the action-type taskblock $TB(do_x^A)$ with $goal(do_x^A) = x$, and do_x^M is the activation condition for the maintain-type taskblock $TB(do_x^M)$ with $goal(do_x^M) = x$.

A maintain-type taskblock will keep a given system condition true, as long as the condition was true initially when the taskblock was activated. This is formally

stated as follows:

Definition 6.4 Given a target condition x from the system, a taskblock with activation signal do_x^M is a MAINTAIN-TYPE TASKBLOCK for x if:

1. $C_{init}(do_x^M) = \{x\}$
2. $goal(do_x^M) = x$

Action-type taskblocks are intended to drive a component to an intended goal from any initial state. This is formally stated as follows:

Definition 6.5 Given a target condition x from the component, a taskblock with activation signal do_x^A is an ACTION-TYPE TASKBLOCK for x if

1. $C_{init}(do_x^A) = \emptyset$
2. $goal(do_x^A) = x$.

For the remainder of this chapter, we are assuming that the conditions sequences we consider are TRIM. We refer the reader to chapter 5 for the definition of a trim C-Sequence.

In [HGSA00] we assumed the component models were constrained by the SYSTEM STRUCTURE ASSUMPTION. In this work, we wish to generalize our discussion about taskblocks and render it relatively independent of the nature of the component models. The following assumptions will define the nature of taskblocks for the remainder of this chapter.

Taskblock Structure Assumption (TSA): Given an activation condition, do_x, it's taskblock, $TB(do_x)$, and component model, $G_{compo}(do_x)$, we assume the following:

1. do_x is effective for control for $G_{compo}(do_x)$.
2. $TB(do_x)$ is a state graph.
3. For each place p in $TB(do_x)$ there exists a condition set C such that:

(a) $\forall t \in {}^{(t)}p, \Phi(t) = C$.

(b) If $\Phi(t) \neq do_x$ and $\Phi(t) \neq \neg do_x$, then $\forall c \in C_{out}(C_{compo}(do_x))$, either $c \in C$ or $\neg c \in C$.

4. Except when idle, the state of $TB(do_x)$ always changes in response to expected changes in condition outputs of $G_{compo}(do_x)$.

5. The state of $TB(do_x)$ does not change in response to unexpected change in conditions output of $G_{compo}(do_x)$.

Here we are assuming that we are effective for control (item 1), and that the model within contains only one marked place at any time (item 2). Item 3 part (a) insures that each state corresponds to some specific output of the plant (do_x, $\neg do_x$ also work here). Item 3 part (b) insures that each transition that inputs to a place in the taskblock has a full condition mapping from the component model. Item 4 states that the taskblock always recognizes expected responses from the component model. The implication of item 5 is that the original taskblock $TB(do_x)$ doesn't respond to unexpected output changes from $G_{compo}(do_x)$, and thus we have an opportunity to add fault detection by adding recognition of these unexpected output changes.

Another key point of the TSA, is that for every state in the taskblock that provides stimuli (via do_x conditions) to the component model, then the taskblock encodes all possible expected responses from the system that lead the the completion of a task.

We are now ready to discuss the general behavior and model characteristics of our taskblocks. While the Maintain and Action taskblocks perform separate functions, they behave in a very similar manner. We note, that a taskblock drives a system component to some specific state when the activation condition do_x is output for every step in the execution of the task.

For fault detection, we will assume that under execution of some task do_x will be output continually. This will simplify our discussion. Basically, taskblocks provide stimuli to the component to drive it to some specific target state, and the taskblock

model then waits for responses from the system (i.e. output conditions) that trigger a state change in the task model. This in turn triggers a new stimuli to the system, and the component's response. In this way our taskblocks are intended to unfailingly guide the component model to this target state.

Figure 6.3 illustrates the behavior of a taskblock under the assumption that do_x is continually input to the system. In essence , a taskblock provides stimuli to the system and waits for appropriate responses from the system to determine the next action (i.e. - stimuli from the controller) to take.

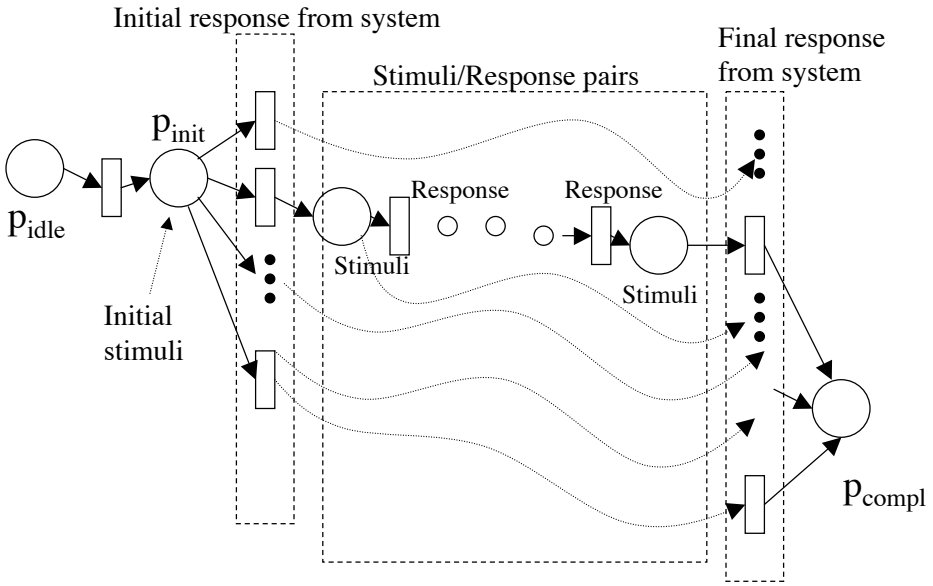

Figure 6.3: An overview of taskblock behavior under a continuous do_x input condition.

We note that by figure 6.3, we have characterized p_{init} as the first stimuli of the taskblock. Its purpose is to allow the taskblock to determine the proper initial output of the component model (as captured by its output conditions), and so while we characterize it as a stimuli, it actually outputs no conditions to the system (i.e. an empty set of output conditions).

6.3 A Fault Detection Scheme for Taskblocks

In this section, we outline a method to modify a taskblock that meets the taskblock structure assumption to include the ability to detect deviations between the real and expected behavior of the component model. Figure 6.4, shows a simple block diagram that outlines our method of transformation.

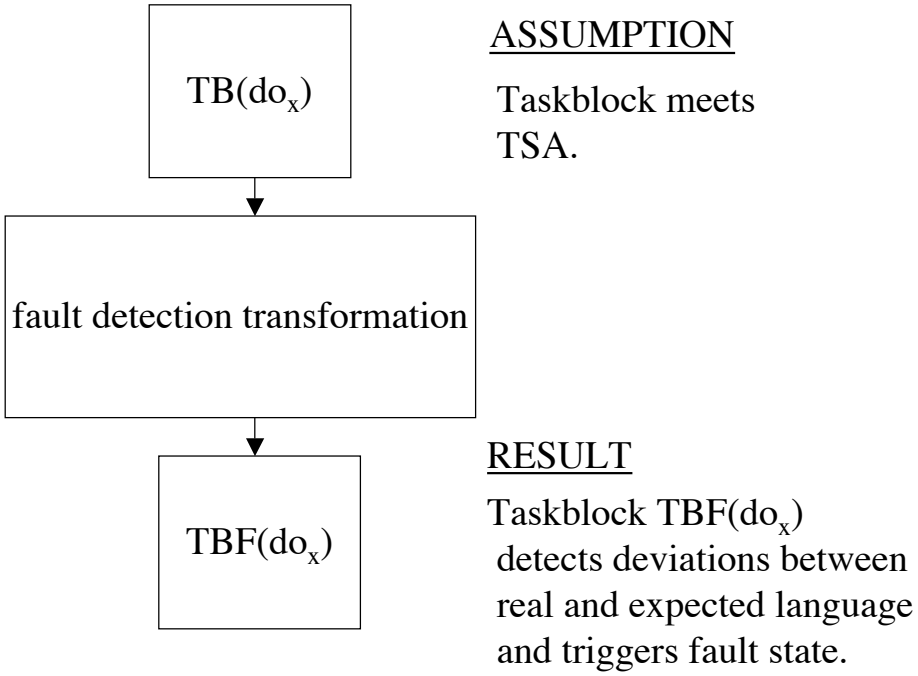

Figure 6.4: The transformation of a $\mathrm{TB}(\mathrm{do}_x)$ into a taskblock that detects faults $\mathrm{TBF}(\mathrm{do}_x)$.

We propose to modify the structure of the taskblock as shown in figure 6.5. We have added a new output condition to the taskblock that outputs a fault condition, $\mathrm{fault}(\mathrm{do}_x)$.

The following defines for the whole system the notion of EFFECTIVE FOR DETECTION. In essence, it says any unexpected stimuli from the system (under direction of a taskblock), leads to a fault state.

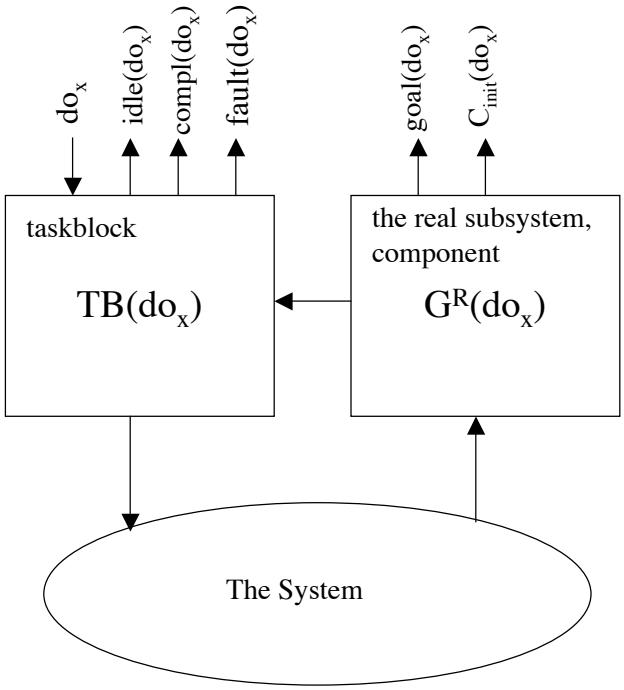

Figure 6.5: The relationship between a taskblock with Fault Detection, its associated real subsystem, and the direct translator.

Definition 6.6 Given a system $\mathcal{G}^E \subseteq \mathcal{G}_{tasks} \cup \mathcal{G}^E_{compo}$ with initial state m^E_0 and a condition $do_x \in C_{in}(\mathcal{G}^E) \cap C_{do_x}$ such that $idle(do_x) \in g(m^E_0)$, do_x is EFFECTIVE FOR DETECTION for $\mathcal{G}^E$ under m^E_0 if :

UNEXPECTED COMPONENT BEHAVIOR WHILE TASKBLOCK ACTIVATED IMPLIES EVENTUAL FAULT: Given any $(C_1 \cdots C_k C_{k+1} C_{k+2})$ such that:

1. Observations are in the expected language up to the k'th observation from the component, or $(C_1 \cdots C_k) \in L(\mathcal{G}^E, m^E_0)$,

2. At the $k+1$ observation of the component, an unexpected behavior is encountered, or $(C_1 \cdots C_k C_{k+1}) \notin L(\mathcal{G}^E, m^E_0)$, and

3. Under continual activation and a faulty behavior, the language generated by the taskblock remains within its expected language, or $(C_1 \cdots C_k C_{k+1} C_{K+2}) \in$

$L(\mathcal{G}_{tasks}, m_0^E)$ and $(\emptyset\{do_x, C_{init}(do_x)\}\{do_x\}) \leq (C_1 \cdots C_k C_{k+1} C_{k+2})$;

then, THE TASKBLOCK MOVES TO A FAULT STATE , or

$(\emptyset\{fault(do_x)\}) \leq (C_1 \cdots C_k C_{k+1} C_{k+2})$.

Item 3 states that any sequence generated by the taskblock remains within the expected language of the taskblock model. This does not mean that the system (in its entirety) stay within it's expected language, but instead that movement to the fault state, p_{fault}, is a legal behavior.

While this definition applies to the whole system, we note that in this initial work we will show that a single taskblock and component model pair are effective for control under the assumptions we have presented.

Define ${}^{(t)}p$ as the set of transitions that are inputs to place p, and $p^{(t)}$ as the set of transitions leading from place p (i.e. output transitions).

The following defines the set of condition sets which represent unexpected behaviors from the component for each place within a taskblock that provides stimuli to the component. This definition is used in algorithm 6.3.

Definition 6.7 Given a taskblock $TB(do_x)$ define for all $p \in \mathcal{P}_{TBF(do_x)} - \{p_{idle}, p_{compl}\}$, the set $CSet_{fault}(p)$ as:

$$CSet_{fault}(p) := 2^{\mathcal{C}_{G_{compo(do_x)}}} - (\cup_{t \in p^{(t)}, {}^{(t)}p} \Phi(t))$$

This is the set of all possible combinations of output conditions minus output conditions that were either expected (i.e. they triggered a state change in $TB(do_x)$ that led to state p being marked), and the ones that are expected (i.e. legal responses given stimuli provided by p).

We are now ready to present the algorithm that we will use to transform $TB(do_x)$ into a new taskblock that detects faults. We will show in the result for this chapter that we do in fact detects deviations between expected and real behavior, and that the input/output behavior of the original taskblock is preserved.

Algorithm 2: An algorithm to create $TBF(do_x)$ from $TB(do_x)$.
Input: do_x, $TB(do_x)$, $G_{compo}(do_x)$
Output: $TBF(do_x)$

(1) Create a place, p_{fault}, Assign output condition $fault(do_x)$ to this place.
(2) $\forall p \in \mathcal{P}_{TBF(do_x)} - \{p_{idle}, p_{compl}, p_{fault}\}$
(3) {
(4) $\forall C \in CSet_{fault}(p)$
(5) {
(6) Add a new transition, t', with C as the enabling
(7) condition for this transition, let the input place
(8) to t' be p, and the ouput place is p_{fault}.
(9) }
(10) }

Note the places, p_{idle} and p_{compl} associated with the status conditions $idle(do_x)$ and $compl(do_x)$ do not interact with the system and hence are excluded from consideration in this algorithm, as is the newly created p_{fault} state.

The algorithm adds, for all places that provide stimuli to the component, transitions leading to the fault state. One of these transitions is added for each combination of conditions that are not expected given the current stimuli. We note that if the component model, $G_{compo}(do_x)$, has n output conditions, then each place that provides stimuli in the new $TBF(do_x)$ will have approximately 2^n transitions added. This is obviously an issue, but we believe we can exploit simple and well known ideas to greatly reduce the number of transitions considered (i.e. Karnaugh mapping reduction). We do not consider this in this chapter. It is a subject of future research.

Figure 6.6 graphically shows what we do for each place (excluding p_{idle} and p_{compl}) to create the ability to perform fault detection.

The following result shows that our new taskblock that detects fault behaviors

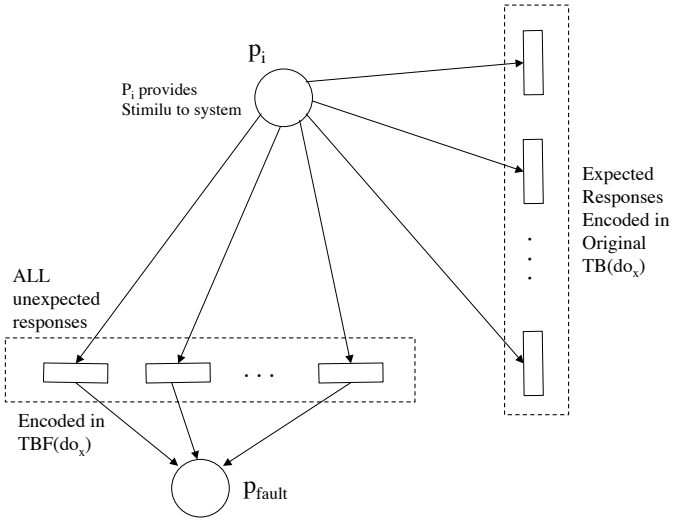

Figure 6.6: A figure explaining how fault detection is encoded into $TBF(do_x)$ one stimuli state at a time.

is effective for control and effective for detection.

Theorem 6.1 Given do_x, a taskblock $TB(do_x)$ satisfying the taskblock stucture assumption(TSA), the component model $G_{compo}(do_x)$, and a modified $TBF(do_x)$ constructed via algorithm 6.3, do_x is EFFECTIVE FOR CONTROL and EFFECTIVE FOR DETECTION.

Proof: By the taskblock structure assumption(TSA) item 3(a), for any place in $TB(do_x)$ all transitions into that place($\forall t \in^{(t)} p$) have the same condition map for the system, so by item 3(b) unless the transition relates to activation or inactivation, there is a complete mapping from conditions from the plant , and only for that mapping will the place become marked. The place place p in $TB(do_x)$ thus corresponds to a set of markings in the component which have the same observation. (Note that this might not correspond to all markings corresponding to the same observation). Call this set of markings for this place as M.

By TSA item 4, any expected change of observations has a corresponding state change in the $TB(do_x)$, so there must be a transition for leaving the place p for any expected change of observations (corresponding to an expected movement from

plant marking set M to some observationally different marking set M').

By the definition of $CSet_{fault}(p)$, there will now also be a new transition leaving the place p corresponding to all observation changes that were not in the original net. These transitions have condition mappings corresponding to unexpected observation changes.

In the $TBF(do_x)$ the original transitions are preserved from $TB(do_x)$, and since all added transitions from $CSet_{fault}(p)$ are distinct, then the taskblock, $TBF(do_x)$,under expected observations will operate identically as before, and so will be EFFECTIVE FOR CONTROL.

Since each of the newly added transitions in $TBF(do_x)$ (corresponding to $CSet_{fault}(p)$) lead to the p_{fault} state, and since $CSet_{fault}(p)$ corresponds only to unexpected condition changes (corresponding to unexpected marking changes from the set M), then it follows that any unexpected behavior while the system is activated will lead to the outputting of $fault(do_x)$. Hence do_x is EFFECTIVE FOR DETECTION. □□□

The taskblock $TBF(do_x)$ contains the original taskblock with the addition of transitions that capture unexpected responses for each stimuli provided by $TBF(do_x)$. We note that the states that provide stimuli from $TBF(do_x)$ are identical to those of $TB(do_x)$(with the addition of the fault state p_{fault} which does not provide stimuli to $G_{compo}(do_x)$). And so, the only way that $TBF(do_x)$ does not reach the completion state (which of course satisfies effective for control) is if one of the unexpected responses leads the system to the fault state. In this way, given some stimuli for a place in the taskblock, we capture all responses from the system including all possible unexpected behaviors.

Example 6.2 Consider p_{init} and p_8 from figure 6.2 as shown in figure 6.7. This figure shows the expansion of these places that occur to construct $TBF(do_x)$ from $TB(do_x)$. To make the figure easier to read, we represent the input conditions arm_up as up, and arm_dn as dn. Note that since we have 3 output conditions from the component model, we have up to $2^3 - 1 = 7$ transitions leaving each of

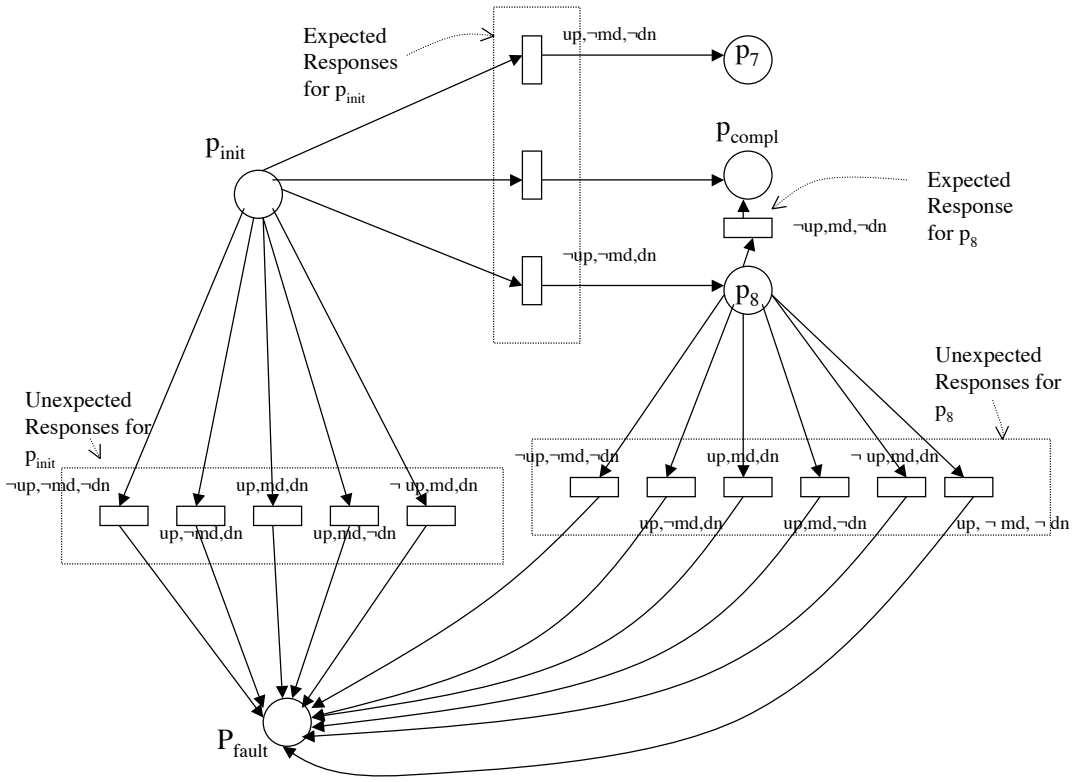

Figure 6.7: The transformation of p_{init} and p_8 from figure 6.2 to the new places in $TBF(do_x)$.

these places (incoming transition conditions are excluded in the set of fault transitions).

6.4 Discussion

In this initial chapter we have shown how to create a taskblock that detects fault behaviors from the system under control. We start with a taskblock satisfying the taskblock structure assumption, and transform it into a new taskblock that is both effective for control and effective for detection. It complements the work of [HGSA00] rather nicely.

We have also included a generalized discussion of the nature of the condition system models representing taskblock structures. We hope this makes these ideas more accessible, while at the same time simplifying our discussion of the synthesis of taskblocks with fault detection. This work has also provided some insight into ways that we may expand the modeling power of the models we use to create component models. Unlike the taskblock which can be synthesized automatically, the component model must be generated by a human. By using higher level modeling structures, this task can be simplified. Also, we do not require that faulty behaviors be encoded within the component model, instead we automatically determine these by the $\mathrm{TBF}(\mathrm{do}_x)$ transformation once again simplifying the modelers job. [1]

We envision that a method such as this could be used in conjunction with a diagnosis method such as the one presented in chapter 7 to provide a method to rapidly detect and diagnose faulty behaviors. Areas of future research include: determining a method to reduce the number of transitions in $\mathrm{TBF}(\mathrm{do}_x)$; finding what types of component models lend themselves to taskblock synthesis; and introducing timing into this framework.

[1] However, note that faults can be explicitly represented in our models. This is done by linking the net representing the deviated part of the behavior to the nominal model with a transition. This transition would have an input condition that would always be assumed to be false. Determining that the condition was not false as was expected/assumed would thus explain the observed faulty behavior.

Copyright © Jeffrey Ashley 2004

Chapter 7

The Diagnostic Causal Network

In this chapter we present a causal network structure in the spirit of the symbolic causal network of [DP94a], [DP94b] [PC98]. In our causal structure, the nodes map to unique variables where each variable is associated with the health or correctness of output of a component. A method to generate a causal structure and associated database of clauses is presented.

A diagnostic causal network (DCN) is a symbolic causal network restricted to describing the interactions of subsystems within the plant. In a DCN each node in the causal structure represents either: 1) the health of a subsystem; or 2) whether a subsystem is outputting the appropriate signal given an expected output signal. An example DCN is shown in figure 7.1 The directed arcs connecting the nodes show the direct influences of subsystems on other subsystems within the whole system, and the terminal nodes represent assumptions about the health of the different subsystems. The DCN can be used to determine a superset of diagnoses corresponding to subsystems for which inconsistent behavior of the subsystem could account for the observed faulty behavior of the entire system.

For any G_i^E, define $Causes(G_i^E) = \{G_j^E \mid C_{out}(G_j^E) \cap C_{in}(G_i^E) \neq \emptyset\}$. This is the set of component models that output a condition that is also an input to the component model in question. From this definition we will build a diagnostic causal network that will allow us to perform our diagnosis. The DCN is a graph where each node in the graph has an associated propositional function, or CLAUSE, built from VARIABLES. The graph serves as a visualization tool, and by its construction we are insured that the subsequent analysis is logically correct [DP94a]. We note that we utilize standard propositional logic syntax in the following discussion, vari-

ables have a truth value of TRUE or FALSE, and $\rightarrow$ represents implication. For each subsystem, $G_i^E \in \mathcal{G}^E$, we define two variables, denoted $CN(G_i^E)$ and $OK(G_i^E)$. Informally, $CN(G_i^E)$ is TRUE if the subsystem is outputting expected conditions, and FALSE otherwise. It therefore represents whether a subsystem is behaving as expected or not. $OK(G_i^E)$ is TRUE if the subsystem is not broken (i.e. its output is consistent with its expected output), and FALSE is if it is broken. We note that if a subsystem is broken then it will not behave as expected.

Definition 7.1 A DIAGNOSTIC CAUSAL NETWORK is a tuple, $DCN = (V, A, E, \Delta)$ where

1. nodes in the causal structure map one-to-one to elements in $V \cup A$;
2. V is a set of nodes (variables) corresponding to each $CN(G_i^E)$ for $1 \leq i \leq n_E$;
3. A is a set of nodes (variables) corresponding to each $OK(G_i^E)$ for $1 \leq i \leq n_E$;
4. $E \subseteq (V \cup A) \times V$ is the set of directed edges connecting nodes. Given some subsystem, G_i^E, the edge $(CN(G_j^E), CN(G_i^E))$ is in E if $G_j^E \in Causes(G_i^E)$, and $(OK(G_j^E), CN(G_i^E))$ is in E if $i = j$;
5. The Δ is a mapping of variables in V to clauses. In particular, the clause $\Delta(CN(G_i^E))$ is the following expression:

$$OK(G_i^E) \wedge (\bigwedge_{G_j^E \in Causes(G_i^E)} CN(G_j^E)) \rightarrow CN(G_i^E);$$

To simplify discussion we simply refer to the nodes in the DCN by their variable name. In the causal net literature, the $OK(G_i^E)$ variables (associated with set A) are called ASSUMABLES [PC98]. Note that for each variable on the left hand side of a Δ clause, there exists an edge in the graph connecting that variable to the variable on the right hand side of the clause. From our clauses, we note that if $CN(G_i^E)$ is FALSE, then some variable (node) that inputs to it must also be FALSE.

Example 7.1 Figure 7.1 represents the causal structures for our example of figure 3.1. We note that all the terminal nodes are assumable nodes that represent the health of a subsystem.

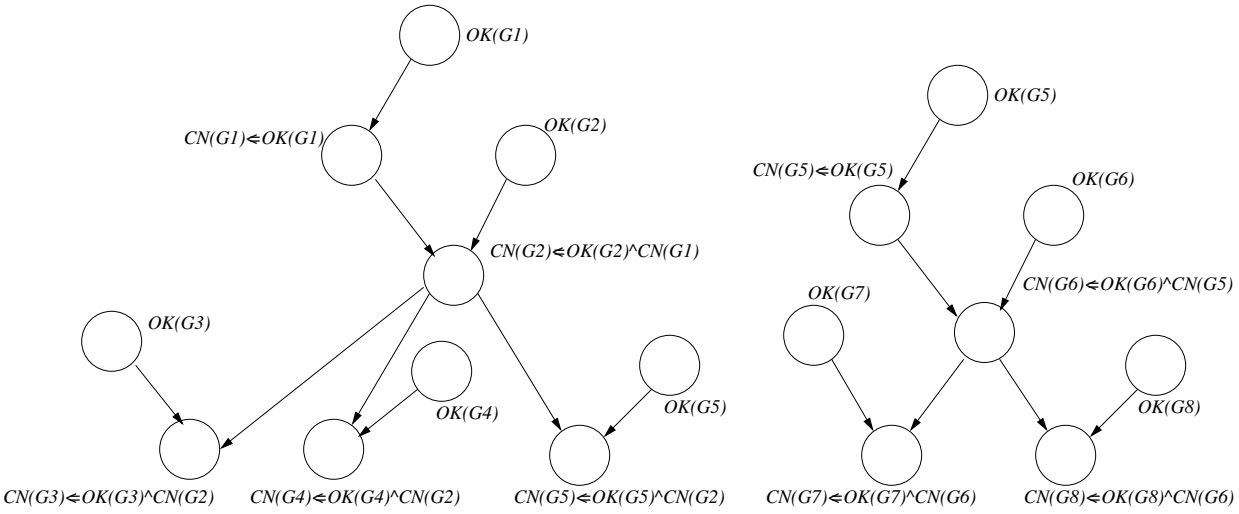

Figure 7.1: A DCN for the condition system shown in figure 3.1.

The following lemma follows directly from the definition of the clauses in the DCN and their relationship to the DCN structure.

Lemma 7.1 Given a causal net without cycles and given a node in $CN(G_i^E)$, if $CN(G_i^E) = FALSE$, then there exists some node $OK(G_j^E)$ such that there is a path from $OK(G_j^E)$ to $CN(G_i^E)$ in the DCN and $OK(G_j^E) = FALSE$.

An interpretation of the DCN clause for a node $CN(G_i^E)$ is as follows: If all inputs to a node are consistent with expected behavior (i.e. the output conditions of $G_j^E \in Causes(G_i^E)$ are within expected behavior), and if that node's model is correct with respect to the real system behavior ($OK(G_j^E)$ is TRUE), then the condition outputs, $C_{out}(G_i^E)$, of this node's subsystem should also be consistent with the expected behavior. Diagnosis then comes in negating this logic. If some observed condition is not consistent with expected operation, then $CN(G_i^E)$ is false for the G_i^E which outputs the condition, and from the DCN clauses, this implies some subsystem which influences (directly or indirectly) this subsystem must have $OK(G_j^E)$ false.

From the lemma above, we can then define this set of influencing systems as the set $RootCauses(c)$: Given a condition c and a subsystem G_i^E such that $c \in C_{out}(G_i^E)$,

define:

$$\begin{aligned}\mathit{RootCauses}(c) \;=\; \{G_j^E \mid &\text{ there exists a path in the}\\ &\text{DCN from } OK(G_j^E) \text{ to } CN(G_i^E),\\ &\text{where } G_i^E \text{ outputs } c\}.\end{aligned}$$

Definition 7.2 Consider an observed C-sequence $(C_0^R \ldots C_{k-1}^R C_k^R)$ such that $(C_0^R \ldots C_{k-1}^R C_k^R) \notin L(\mathcal{G}^E, m_0^E)|_{C_{obs}}$ but where $(C_0^R \ldots C_{k-1}^R) \in L(\mathcal{G}^E, m_0^E)|_{C_{obs}}$. Define $CSets^{Bad} \subseteq 2^{AllC}$ such that for a given set C^{Bad}, then $C^{Bad} \in CSets^{Bad}$ if:

1. $C^{Bad} \subseteq C_k^R$;
2. $C^{Bad} \not\subseteq C_k^E$ for any C_k^E for which $(C_0^R \ldots C_{k-1}^R C_k^E) \in L(\mathcal{G}^E, m_0^E)|_{C_{obs}}$;
3. there is not a strict subset of C^{Bad} satisfying items 1 and 2 above.

C^{Bad} is then a set of observed conditions which indicate that a fault has occurred.

Example 7.2 Suppose that our language $L(\mathcal{G}^E, m_0^E)$ includes the following strings:

$$\begin{aligned}s_1 &= (\{c_1, c_2, c_3\}\{c_1, c_2, c_3\})\\ s_2 &= (\{c_1, c_2, c_3\}\{\neg c_1, c_2, c_3\})\\ s_3 &= (\{c_1, c_2, c_3\}\{c_1, \neg c_2, \neg c_3\})\\ s_4 &= (\{c_1, c_2, c_3\}\{\neg c_1, \neg c_2, \neg c_3\})\end{aligned}$$

If we observe $s_{obs} = (\{c_1, c_2, c_3\}\{\neg c_1, c_2, \neg c_3\}) \notin L(\mathcal{G}^E, m_0^E)$, $CSets^{Bad}$ consists only of the set $C^{Bad} = \{c_2, \neg c_3\}$. Note that out of all continuation steps in the language, none have the $\{c_2, \neg c_3\}$ as an allowable subset, and there is no smaller subset of this that could also satisfy the requirements in definition 7.2; any strict subset of this (such as $\{c_2\}$ or $\{\neg c_3\}$) would be a subset of some C_k^E.

If $L(\mathcal{G}^E, m_0^E)$ contained s_1 and s_4 but not s_2 and not s_3 (e.g. $c_1 = c_2 = c_3$ always), then $CSets^{Bad} = \{\, \{c_2, \neg c_3\}, \{\neg c_1, c_2\} \,\}$.

We are now ready to state the second result of this chapter.

Theorem 7.1 Given an observed $s_{obs} = (C_0^R \ldots C_{k-1}^R C_k^R) \notin L(\mathcal{G}^E, m_0^E)$ with $CSets^{Bad}$ as defined in definition 7.2, if the DCN is without cycles, then

$$Diagnosis(s_{obs}) \subseteq \bigcap_{C^{Bad} \in CSets^{Bad}} \left(\bigcup_{c' \in C^{Bad}} Rootcauses(c') \right) \tag{7.1}$$

Proof: From our definition 7.2, $C^{Bad} \in CSets^{Bad}$ is a set of conditions in C_k^R that triggered our fault detection. From the definition, for some $c \in C^{Bad}$ (where c may be a negated condition), either:

1. c should not have become true following any modeled behavior matching $(C_0^R \ldots C_{k-1}^R)$, or

2. there is some subset $C \subseteq C^{Bad}$ such that all conditions in $\{c\} \cup C$ should not be true simultaneously following any modeled behavior matching $(C_0^R \ldots C_{k-1}^R)$.

(Note case 1 is a special case of case 2 when $C = \emptyset$.) In either case by the MRR assumption, the violated behavior restriction must correspond to some subsystem $G_i^E \in Diagnosis(s_{obs})$, where (from definition 4.2) s_{obs} is in $L(\mathcal{G}^E/G_i^E, m_0^E)|_{C_{obs}}$, the relaxed behavior.

The fact that relaxing the behavioral constraints imposed by G_i^E would allow c means that for some $c' \in C^{Bad}$ (where c' could be c), either $c' \in C_{out}(G_i^E)$, or there exists some sequence of subsystems from G_i^E to some subsystem G_j^E which drives c', where each subsystem in the sequence outputs conditions which input to the next. From the definition of the causal net, that means that the sequence of subsystems corresponds to a path from node $OK(G_i^E)$ to $CN(G_j^E)$, and thus $G_i^E \in RootCauses(c')$ for some $c' \in C^{Bad}$. Under the single fault assumption, for each C^{Bad} set in $CSets^{Bad}$, there exists such a $c' \in C^{Bad}$ such that $G_i^E \in RootCauses(c')$. This gives the intersection in equation 7.1. Thus, the theorem is proved. □□□

We note that the reverse of the inclusion in the theorem is not necessarily true: It is possible to have some G_i^E in $Rootcauses(c')$ but which is not a valid diagnosis. For example, some G_i^E may output a c'' which is an input to a subsystem G_j^E that

then outputs a $c' \in C^{Bad}$, but based on the dynamics within G_j^E under its current possible markings, the value of c'' has no immediate influence on the marking evolution of G_j^E. Thus, relaxing the behavioral constraint imposed by G_i^E will not affect G_j^E, and so G_i^E will not be in $Diagnosis(s)$.

Example 7.3 We continue the example 4.1:

- For observed sequence s_1:

$$\begin{aligned} CSets^{Bad} &= \{\,\{AtLeft, AtHome\}\,\} \\ RootCauses(AtLeft) &= \{G_1, G_2, G_3\} \\ RootCauses(AtHome) &= \{G_1, G_2, G_4\} \end{aligned}$$

  From the theorem, we have $Diagnosis(s_1) \subseteq \{G_1, G_2, G_3, G_4\}$, which is true since we earlier determined that $Diagnosis(s_1) = \{G_2, G_4\}$. Note that the diagnostic superset from the theorem includes G_1 and G_3 which are not in $Diagnosis(s_1)$. This is because there is a cause and effect relationship between these subsystems and the observed conflicting conditions, even though neither could be responsible under the single-fault assumption.

- For the observed sequence s_2:

$$\begin{aligned} CSets^{Bad} &= \{\,\{AtLeft, AtHome\}, \{AtLeft, AtRight\}, \\ &\qquad \{AtHome, AtRight\}\,\} \\ RootCauses(AtLeft) &= \{G_1, G_2, G_3\} \\ RootCauses(AtHome) &= \{G_1, G_2, G_4\} \\ RootCauses(AtRight) &= \{G_1, G_2, G_5\} \end{aligned}$$

  From the theorem, we then have:

  $$Diagnosis(s_2) \subseteq \{G_1, G_2\}$$

  In example 4.1, we determined that $Diagnosis(s_2) = \{G_2\}$, which is smaller than the calculated set $\{G_1, G_2\}$. However, this example illustrates how the multiple C^{Bad} sets can be used to narrow a diagnosis.

In this chapter, we have shown how to exploit the causal structure of our condition system representation of the plant to perform a rapid (but not best-possible) fault diagnosis. Conditions (some of which may be unobservable) provide communication between subsystems and we exploit this to determine which subsystems influence other subsystems through conditions. From this we build a causal structure that can be used to rapidly perform a diagnosis given sets of known-bad observable conditions. In future work, we would like to refine the diagnosis by also considering the interior dynamics of our subsystem models.

Copyright © Jeffrey Ashley 2004

Chapter 8

An Equivalent LTL/Kripke structure for the Condition Sequence/Condition System Model

In this chapter, we show that our condition sequence (the specification) and condition system (the model of the system) have an equivalent structure in the temporal logic framework. In particular, we show that there exists a linear-time temporal logic (LTL) specification and a Kripke structure that represents our specification and model. Linear time logic is a propositional logic with several time operators added. These operators allow for the specification of sequentially timed behaviors used in verification.

Temporal logic is of great interest to the computer science community and there is a host of literature covering different flavors of temporal logic, a few of which are CTL, CTL*, and LTL. As stated, temporal logic is largely used in addressing the verification problem. A Kripke structure (a labeled finite state machine) specifies some process, and temporal logic statements can be used to specify desired properties of this process. In verification, this information is used to determine whether the specified properties are satisfied by the Kripke structure. For some good references in temporal logic see [HR00, MP91, Kro87, GO94].

For this paper, we will use LTL because it suits the nature of our specification. Our aim of this work is to show how our research in condition systems is related to temporal logic research, and to better understand current trends in such a related field. We also hope that it will allow us to use some of the ideas found in temporal

logic research in our own work, and to potentially make our research more accessible to researchers in temporal logic. A prior investigation in the use of temporal logic in the field of DES by a collaborator also motivated this investigation. For interesting applications of temporal logic to DES see [JK03b, JK03c, JK03a].

In the remainder of this chapter we show how the condition system and condition sequence map into the LTL framework. In the next section we introduce the LTL notation used in the remainder of the paper. Then we introduce a one way translation from condition sequence to an LTL formulation and a one way translation from condition system to fair Kripke Structure. In the last section we show that the systems are equivalent in terms of determining whether a string belongs to the observed language of the system or not. In particular, we show that a C-Sequence is within the language generated by a condition system if and only if the LTL formula generated by the C-Sequence is satisfied by the Kripke Structure generated from the Condition System. We use a simple example throughout to display the concepts presented.

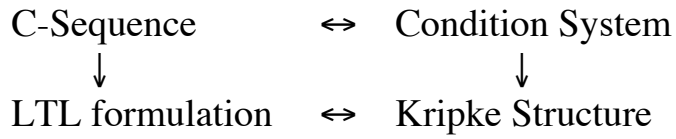

Figure 8.1: An overview of the relationship between the Condition Sequence/System and the LTL framework.

8.1 Linear-time temporal logic.

Typically linear-time temporal logic is used to specify qualitative properties (desired or otherwise) that can be used in conjunction with a discrete model of some system to evaluate whether the properties will hold for the system as it changes over time [MP91] . In this logic system, quantitative timing is ignored which allows for very general statements about the functioning of such systems. This is the formal verification problem that is prevalent in software and systems research. It is particularly

important in the analysis of reactive and concurrent systems. In verification, two techniques are utilized for problem resolution, automatic theorem proving techniques and model checking. In a manner similar to much DES research, the model checking process utilizes state machine structures in the problem solving process. This is directly analogous method to our method of specifying ambiguous desired behaviors of a system (modeled by a condition system) using condition sequences.

In this section we formally define the linear-time temporal logic syntax that we will use throughout the remainder of this chapter. We adopt the notation of [HR00] for the linear-time temporal logic and a Kripke structure. While subsequent research in temporal logic has extended the logic and temporal operators defined, the set of temporal and propositional operators presented (which is missing $\vee$, $\rightarrow$,$\leftrightarrow$, and $\mathcal{B}efore$) is sufficient for the translation from C-sequences to LTL formulas.

In definition 8.1 we define a Kripke structure, and in definitions 8.2 and 8.3 we define the LTL syntax and an evaluation respectively. Let AP be a set of atomic propositions (conditions) which have a value of TRUE or FALSE.

Definition 8.1 A Kripke structure $\mathcal{M} := \{S, E, \mathcal{L}abel, s_0\}$ is a discrete model where S is a set of states, $E \subseteq S \times S$ is a transition relation, a labeling function $\mathcal{L}abel : S \rightarrow 2^{AP}$ that maps states of the system to atomic propositions, and an initial marking $s_0 \in S$ such that every $s \in S$ has some $s' \in S$ such that $(s, s') \in E$(liveness property).

The temporal operators used in LTL allow for the timed specification of properties where these operators have the following intuitive meaning:

1. $\mathcal{U}$: The operator $\mathcal{U}$ represents the until operation between two propositions. Informally, $p\mathcal{U}q$ means that proposition p is true until proposition q becomes true.

2. G : The G operator represents "globally" or "always", in that Gp represents the specification that p is true for all time.

3. F : The F operator represents "eventually" or "in the future", in that Fp represents the specification that p will hold sometime in the future.

4. X : The next state operator has the meaning for Xp that p will hold in the next state.

Given these temporal operators, the following formally defines the linear-time temporal logic we will use for this chapter. We use the notation from [HR00].

Definition 8.2 Linear-time temporal logic (LTL) has the following syntax given in Backus Naur form:

$$\phi ::= p \mid (\neg\phi) \mid (\phi \wedge \phi) \mid (\phi U \phi) \mid (G\phi) \mid (F\phi) \mid (X\phi),$$

where p is any atomic proposition.

An LTL formula is evaluated over a PATH or a set of paths (also called a STATE TRACE) in the Kripke structure, where a path is $\pi := s_0, s_1, \cdots, s_n$ for some integer $n \geq 0$ where each s_i for $0 \leq i \leq n$ is a element of S. Define $\pi(i) := s_i, \cdots, s_n$ for some integer $0 \leq i \leq n$. These paths represent a qualitative time line of propositional valuations which as we shall see is directly comparable to our C-sequence. With a slight abuse of notation we will represent some path π that exists in $\mathcal{M}$ as $\pi \in \mathcal{M}$.

The satisfaction relation as defined below shows how a Kripke structure and a LTL formulation are related in the verification process.

Definition 8.3 A path $\pi := (s_0, s_1, \cdots, s_n) \in \mathcal{M}$ SATISFIES an LTL formula via the satisfaction relation $\models$ for LTL formulas as follows:

1. $\pi \models \top$, or $true$.
2. $\pi \models p$ IFF $p \in Label(s_0)$.
3. $\pi \models \neg\phi$ IFF $\pi \not\models \phi$.
4. $\pi \models \phi_1 \wedge \phi_2$ IFF $\pi \models \phi_1$ and $\pi \models \phi_2$.
5. $\pi \models \phi_1 U \phi_2$ holds IFF there is some $i \geq 0$ such that $\pi(i) \models \phi_2$ and for all $j = 0, \ldots, i-1$ we have $\pi(i) \models \phi_1$.

6. $\pi \models G\phi$ holds IFF , for all $i \geq 0, \pi(i) \models \phi$.

7. $\pi \models F\phi$ holds IFF , for some $i \geq 0, \pi(i) \models \phi$.

8. $\pi \models X\phi$ IFF $\pi(1) \models \phi$.

Example 8.1 To illustrate these definitions consider the Kripke System shown in figure 8.2. The Set of $Labels$ is $\{off, on, up, mid, down, atUp, atDn\}$. The initial state is the top right node in the figure. To simplify the graph, we have listed only the non-negated conditions for each node.

A simple LTL formulation is $(up \wedge atUp)U(\top U(on \wedge atDn))$.

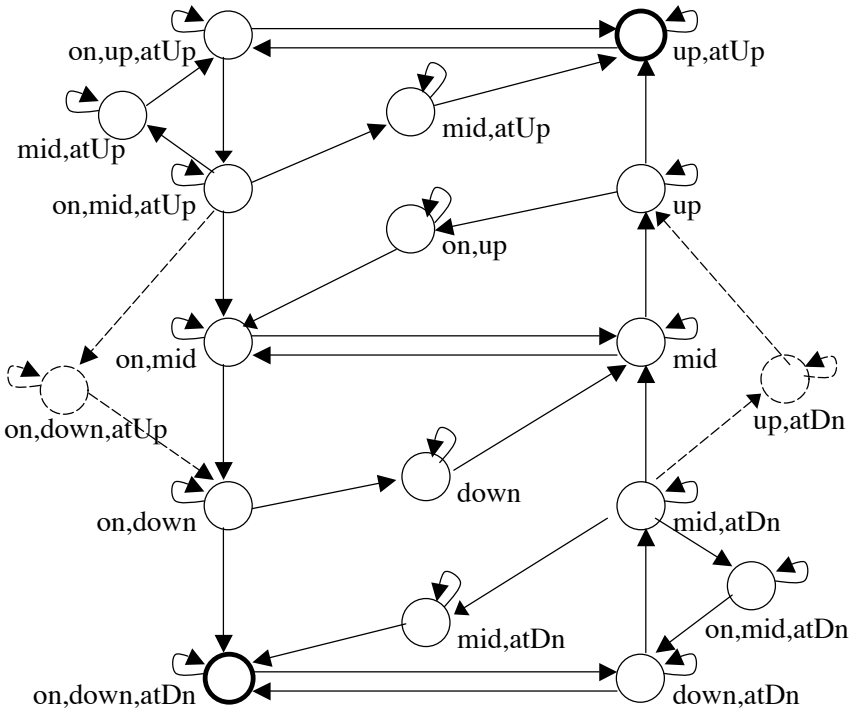

Figure 8.2: The Kripke Structure used in example 8.1

8.2 A translation from the C-Sequence to the LTL formula framework

In this section, we present definitions that define a transform from our model and specification (the condition system and condition sequence respectively) into a Kripke structure and a LTL formula. In the future, this will allow us to draw comparisons between both approaches and potentially extend our research to include results from the temporal logic community. Since the research in temporal logic is quite extensive and well known, we also hope that this will make our work more accessible to the research community by presenting our system description in this manner.

One problem lies in the fact that a condition system model of a plant is an input output model where inputs to the system (i.e.- the control decisions) are considered part of $L(\mathcal{G})$, and a Kripke structure is a labeled state machine with no equivalent provision for system input. So we will first modify our condition system model of the plant by adding models that represent the inputs to the system which will convert the system into an output model where inputs are unconstrained. We will then show that these models are equivalent in terms of language analysis.

Define $C_{in} := (C_{in}(\mathcal{G})/C_{out}(\mathcal{G}))\ |_{C_{obs}}$ as the set of all input conditions to the system. These are the conditions that are input, not outputs and are observable. For the remainder of this chapter we will assume that the values of input conditions to a system $\mathcal{G}$ are known at time 0 of the plant(i.e. at the initial marking m_0). Thus define $C_{in}^0 \subseteq C_{in}$ as the set of input conditions to the system that are $true$ at time 0. Given this definition, define the following language.

$L^0(C_{in}^0) := \{s = (C_0C_1 \dots C_n)$ such that $\forall c \in C_{in}$, if $c \in C_{in}^0$, then $c \in C_0$ and otherwise $\neg c \in C_0$.

This language represents all possible C-sequences given only the constraint that the initial condition set in the sequence is consistent with the initial inputs to the system. This language is used in lemma 8.1. We note that this language need not be determined (and in fact usually cannot as the set is typically infinite), but determining if a sequence is in the language reduces to a simple set comparison of

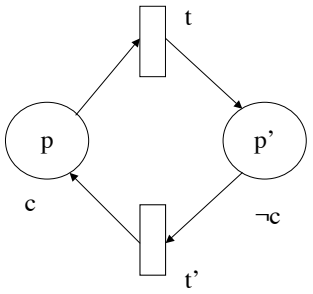

Figure 8.3: A subsystem added to $\mathcal{G}$ to create $\mathcal{G}_{\mathcal{MOD}}$.

C_0 and C_{in}^0.

The following defines our modified condition system model where each input condition to the system is represented in a new model that is appended to the existing system model $\mathcal{G}$. See figure 8.3 for a graphical description of one these new models.

Definition 8.4 Given $\mathcal{G}$, m_0, and a set of conditions C_{in}^0 without contradictions, define $\mathcal{G}_{\mathcal{MOD}}$ as a condition system composed of $\mathcal{G}$ and new subsystems, $G_{c,\mathcal{MOD}}$ (as shown in figure 8.3) for each condition $c \in C_{in}$. Each of these subsystems have the following properties.

1. There are two places (p, and p') such that $\Phi_{\mathcal{G}}(p) = \{c\}$ and $\Phi_{\mathcal{G}}(p') = \{\neg c\}$.
2. There are two transitions (t and t') such that $\Phi_{\mathcal{G}}(t) = \Phi_{\mathcal{G}}(t') = \emptyset$.
3. These components are connected with edges to create a cyclic structure as shown in figure 8.3.
4. Assign one token to one of these places using the following rule: if $c \in C_{in}^0$ then $m(p) = 1$ else $m(p') = 1$.

We need to do this to insure that we capture the input conditions of the system in the Kripke model. In the normal operation of a condition system G, some other entity (i.e.-a controller) inputs these signals to the system, but in our analysis

(which is independent of controller input) these signals are undefined. This modification will allow us to convert the system into our state-based Kripke system, by in effect converting input conditions into mappable states.

The following lemma shows that in terms of language analysis, that $\mathcal{G}$ and $G_{i,\mathcal{MOD}}$ are equivalent.

Lemma 8.1 Given $\mathcal{G}$, m_0, C^0_{in} and a corresponding $\mathcal{G}_{\mathcal{MOD}}$ and $m_{0,\mathcal{MOD}}$ generated from application of definition 8.4, then $s \in L(\mathcal{G}, m_0) \cap L^0(C^0_{in})$ if and only if $s \in L(\mathcal{G}_{\mathcal{MOD}}, m_{0,\mathcal{MOD}})$.

Proof: Let $\mathcal{G}'$ represent the set of subsystems created in Definition 8.4 for each $c \in C^0_{in}$, and m' its corresponding marking for the set of subsystems. Then $L(\mathcal{G}', m') = L^0(C^0_{in})$. By Theorem 3.1, since $\mathcal{G}_{mod} = \mathcal{G} \cup \mathcal{G}'$, then $L(\mathcal{G}_{\mathcal{MOD}}, m_{0,\mathcal{MOD}}) = L(\mathcal{G}, m_0) \cap L(\mathcal{G}', m'_0)$, which thus equals $L(\mathcal{G}, m_0) \cap L^0(C^0_{in})$. □□□

We also need to define immediate reachability for condition systems to implement our condition system to Kripke structure translation definition. Note that this definition differs from definition 5.2 in that it is defined over the marking space independent of conditions sets, whereas definition 5.2 is defined over markings and observable condition sets. Definition 8.5 represents the standard definition of immediately reachability extended to condition systems.

Definition 8.5 Define the IMMEDIATELY REACHABLE MARKING SET $R^1(m)$ for some marking m as the set of all immediately reachable markings :

$R^1(m) := \{m' \mid m' \in f_{\mathcal{G}}(m, C) \mid \exists C \in 2^{AllC}\}$

We use definition 8.5 in the following definition. In it we define a Kripke structure($\mathcal{K}_{\mathcal{G}}$) generated from enumerating all possible states of $\mathcal{G}_{\mathcal{MOD}}$ given some initial marking. We note that the marking space of $\mathcal{G}_{\mathcal{MOD}}$ will be finite since the marking space ($\mathcal{M}_{\mathcal{G}}$)of $\mathcal{G}$ is finite. Denote the marking space of $\mathcal{G}_{\mathcal{MOD}}$ by $\mathcal{M}_{\mathcal{MOD}}$. We note that the state space of $\mathcal{G}_{\mathcal{MOD}}$ will be $\mathcal{M}_{\mathcal{MOD}} = 2^{|C_{in}|}|\mathcal{M}_{\mathcal{G}}|$ due to the addition of the subsystem models used to represent the input conditions to the system.

Definition 8.6 Given some condition system and initial marking $\mathcal{G}_{\mathcal{MOD}}$ and $m_{0,\mathcal{MOD}}$, define $\mathcal{K}_{\mathcal{G}}$ as a Kripke structure that is created by the following:

1. (a) Let $AP = AllC$.
   (b) Given $\mathcal{G}_{\mathcal{MOD}}$ and $m_{0,\mathcal{MOD}}$ enumerate the marking space and map each marking, m_i to a new node, s_i in the Kripke structure.
   (c) For each node, s_i, assign the Labeling as follows $Label(s_i) = g(m_i)$.
2. For each $(m_i, m_j) \in \mathcal{M}_{\mathcal{MOD}}$ such that $m_j \in R^1(m_i)$, add an edge from s_i to s_j in the corresponding Kripke structure, $(s_i, s_j) \in E$.
3. Starting with the node s_0 corresponding to $m_{0,\mathcal{MOD}}$ find all reachable nodes within the Kripke structure. Remove all nodes from the structure that are unreachable.
4. Assign the initial marking s_0 to the node corresponding to $m_{0,\mathcal{MOD}}$.

Example 8.2 Consider the condition system shown in figure 8.4. This condition system and its initial marking was used to generate the Kripke structure shown in figure 8.2. This system is a modified version of figure 3.1 where the input conditions have been removed. This is directly analogous to the creation of $\mathcal{G}_{\mathcal{MOD}}$ and $m_{0,\mathcal{MOD}}$ and here we have merely reduced the number of subsystems thereby making the example clearer.

In the previous chapter we defined a trim C-Sequence as one of minimal length for its respective equivalence class. We also pointed out that given some C-Sequence we can easily make it trim by removing extraneous condition sets. In the next definition we define a LTL formulation of a C-Sequence.

Definition 8.7 Given some trim C-Sequence , $s = (C_0C_1 \dots C_n)$, define the LTL formula f_s given the following construction rules:

1. If $C_i \neq \emptyset$ let compound proposition ϕ_i be equal to the logical anding of each condition $c \in C_i$(i.e. if $C_i = \{c_1, \neg c_2, c_4\}$ then $\phi_i = (c_1 \wedge \neg c_2 \wedge c_4)$).

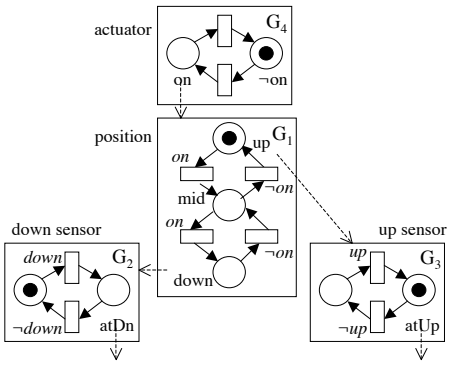

Figure 8.4: The simplified Up/Down Actuator condition system used to generate the Kripke Structure in figure 8.2.

2. If $C_i = \emptyset$ then $\phi_i = \top$.

3. The formula is constructed from these compound propositions in the following manner, $f_s = \phi_1 U(\phi_2 U(\ldots U\phi_n)\ldots)$.

Our condition sequence represents a timeline ordering that is directly related to the UNTIL operator, U. We note that the parentheses are due to the binding precedence of this operator.

Example 8.3 As a continuation of examples 8.1 and 8.2, consider the C-Sequence $s_{obs} = (\{up, atUp\}, \emptyset, \{on, down, atDn\}$.

The equivalent LTL formulation is $(up \wedge atUp)U(\top U(on \wedge down \wedge atDn))$. Which we see is achievable via inspection of figure 8.2.

In the following theorem we show that the LTL/Kripke system is equivalent to its corresponding C-Sequence/condition system in terms of language analysis.

Lemma 8.2 Given a $\mathcal{G}_{\mathcal{MOD}}$, $m_{0,\mathcal{MOD}}$, a corresponding Kripke structure $\mathcal{K}_{\mathcal{G}}$ created by application of definition 8.6, and a trim C-sequence s and its corresponding LTL formula f_s generated by definition 8.7 then the following is true:

$$s \in L(\mathcal{G}_{\mathcal{MOD}}, m_{0,\mathcal{MOD}}) \text{ iff } \exists \text{ some } \pi \in \mathcal{K}_{\mathcal{G}} \text{ s.t. } \pi \models f_s \ .$$

Proof:

(Only If:) Let $s \in L(\mathcal{G}_{\mathcal{MOD}}, m_{0,\mathcal{MOD}})$, we need to show that there exists a $\pi \in \mathcal{K}_{\mathcal{G}}$ such that $\pi \models f_s$. Given that $s = (C_0, C_1, \ldots, C_n)$ is in the language of the system then there exists some marking sequence $(m_{0,\mathcal{MOD}}, m_1, \ldots, m_k)$ for some $k \geq n$ in the system that generates s. Note that there may be more than one marking sequence that generates s. We also note that each marking m_i is immediately reachable from m_{i-1} for $i > 0$ by definition and as a direct result of definition 8.6 we see there exists some $\pi = (s_0, s_1, \ldots, s_k) \in \mathcal{K}_{\mathcal{G}}$. Now consider f_s, we need to show that $\pi \models f_s$. The path π in the Kripke structure must output $(C_0, C_1, \ldots, C_n)$ which is a direct result of definition 8.6. f_s, which was generated by application of definition 8.7 given s, contains each condition set from s joined in order using the Until operator. We see then that the path π satisfies f_s from definition 8.3.

(If:) It suffices to show that given some $s \notin L(\mathcal{G}_{\mathcal{MOD}}, m_{0,\mathcal{MOD}})$ that there does not exist a $\pi \in \mathcal{K}_{\mathcal{G}}$ such that $\pi \models f_s$. Our logic follows the first part of this proof. First, since $s \notin L(\mathcal{G}_{\mathcal{MOD}}, m_{0,\mathcal{MOD}})$, then there can't exist a marking sequence in $\mathcal{G}_{\mathcal{MOD}}$ given $m_{0,\mathcal{MOD}}$. Since no such marking sequence exists then there does not exist a path $\pi \in \mathcal{K}_{\mathcal{G}}$ that outputs $(C_0, C_1, \ldots, C_n)$ by definition 8.6. Hence there exists no path $\pi \in \mathcal{K}_{\mathcal{G}}$ that can satisfy f_s. Thus the lemma is proved. □□□

In this final theorem we extend lemma 8.2 to consider our original condition system $\mathcal{G}$ by using the result lemma 8.1.

Theorem 8.1 Given a $\mathcal{G}$, m_0, and C^0_{in}, then there exists a corresponding Kripke structure $\mathcal{K}_{\mathcal{G}}$ created by application of definitions 8.4 and 8.6, and a trim C-sequence s and its corresponding LTL formula f_s generated by definition 8.7 where the following is true:

$$s \in L(\mathcal{G}, m_0) \cap L^0(C^0_{in}) \text{ iff } \exists \text{ some } \pi \in \mathcal{K}_{\mathcal{G}} \text{ s.t. } \pi \models f_s \ .$$

Proof:

From definition 8.4 we know that we can create a $\mathcal{G}_{\mathcal{MOD}}$ and a new marking $m_{0,\mathcal{MOD}}$ that by lemma 8.1 is known to generate the same language as the

original system. From definition 8.6 we know that $\mathcal{G}_{\mathcal{MOD}}$, and $m_{0,\mathcal{MOD}}$ can be used to generated a Kripke structure $\mathcal{K}_{\mathcal{G}}$. From lemma 8.2, we know that any $s \in L(\mathcal{G}_{\mathcal{MOD}}, m_{0,\mathcal{MOD}})$ has a corresponding f_s generated from definition 8.7 that is satisfied by some path $\pi \in \mathcal{K}_{\mathcal{G}}$. It is obvious then that by chaining these two lemmas together, that given $s \in L(\mathcal{G}, m_0)$ then $s \in L(\mathcal{G}_{\mathcal{MOD}}, m_{0,\mathcal{MOD}})$ and so there exists some $\pi \in \mathcal{K}_{\mathcal{G}}$ that will satisfy f_s. If $s \notin L(\mathcal{G}, m_0)$ then $s \notin L(\mathcal{G}_{\mathcal{MOD}}, m_{0,\mathcal{MOD}})$ and so there will not exist a path $\pi \in \mathcal{K}_{\mathcal{G}}$ that will satisfy f_s. Thus the theorem is proved. □□□

In the future, we would like to show how to convert an LTL formula into a condition sequence and a Kripke structure into a condition system, and to determine under which conditions this is possible. Converting a Kripke structure into a finite condition system is straightforward, but we suspect that conversion of an LTL formulation into a condition sequence is less direct. This is a subject of future research.

Copyright © Jeffrey Ashley 2004

Chapter 9

Discussion

In this dissertation, we explored the the problem of fault detection and fault diagnosis for systems modeled as condition systems. The language of the system is shown to be the intersection of the condition languages defined by each subsystem, and this allowed us to introduce the notion of a relaxed language defined as the intersection of condition languages for all but one subsystem. A system FAULT was defined as an observed behavior which doesn't correspond to any expected behavior. A DIAGNOSIS of this fault localizes the subsystem that is the source of the discrepancy between output and expected observations. A DETECTION is the determination that the system is not behaving as expected according to the model of the system. Critical to this is the notion of the real system and the expected system.

We showed that detection and diagnosis can be determined in a finite number of calculations (i.e. effectively computable). As a result of the proof, which was constructive in nature, we outlined an algorithm to determine the exact solution to the diagnosis problem within the constraints of observability.

The exact diagnosis solution can be computationally involved, so we also introduced a fault detection scheme that modifies our control structure called a taskblock(also a condition system) to implement on-line fault detection. We then showed that the new taskblock preserves the control structure of the original model and does in fact detect unexpected behaviors. We also presented a method to perform a rapid detection and diagnosis that yields a superset that contains the exact diagnosis using a DIAGNOSTIC CAUSAL NETWORK.

We also included a chapter on a conversion from the condition system framework into a linear-time temporal logic (LTL) framework. LTL is principally used in

program and process verification. This is closely related to the diagnosis and detection problem. Our intent of this chapter was to make our work more accessible by tying it into the better known and well investigated LTL framework. It also gave us insight in how the verification problem and the diagnosis problem are related.

Our solutions requires no modeling of faults which saves the system designer the potentially monumental task of specifying this information. They instead utilize existing models used for controller synthesis. This reuse of models improves the cost/benefit ratio of our overall systematic approach to system design. As noted earlier in this dissertation, the modeling of faults can help refine a diagnosis by eliminating obviously contradictory diagnoses. But any approach that requires explicit modeling of faults appears to restrict a diagnosis to these defined fault states (i.e. no information regarding unmodeled faults is included in a diagnosis).

We have developed a Win32 C++ Diagnosis Class and a console program that performs the system diagnosis from chapter 7 in this dissertation. The program takes as input the system model and a $CSets^{Bad}$ set, and returns the set of subsystem names that could be the source of the observed faulty behavior. The Diagnosis Class uses net model files generated by SpecTool [HGSA00] and relies upon SpecTool's NetStructureDLL for access to our existing Petri Net Class. This software is designed to be extendable as this research evolves. Currently, the diagnosis program converts each subsystem model into a causal node, then creates the diagnostic causal model. A diagnosis is then easily performed by direct application of the equation detailed in Theorem 7.1.

Based on our approach, a fault can be created by one of two broad types of occurrences. In the first, a component in the real system breaks thus creating the fault. A more subtle occurrence leading to a fault (by our definition) could be from an inaccurate model of the true dynamics of the system. This idea is very similar to the notion of process verification, and so one area of future research would be in the application of the ideas presented in this dissertation to the problem of process verification. Such faults represent a serious problem because any controller developed to drive a real world plant would be derived from this incorrect model and so could create extremely erratic and potentially dangerous results. Traditional methods of

discrete controller generation (i.e. ad hoc programming) avoid this issue to some degree by implementing various standardized methods and incorporating verification into the design process, but even such techniques provide limited safeguard under an incorrect view of the system (i.e. the specification is invalid).

In other future research, we will analyze the causal structure within the subsystems as well. By considering the current and recent markings of the subsystems, we should be able to determine a smaller set of root causes for a given condition in C^{Bad}, and thus provide a more focused diagnosis. Fault diagnosis given the possibility of multiple failures is another rich area of future research. This is especially important for systems where one fault may cascade into multiple failures. It is anticipated that this will complicate the diagnosis problem significantly.

The detection problem will most certainly generate further opportunities to consider modeling and timing issues among other things. While we have provided some qualitative solutions to the fault detection and diagnosis problem, it seems that any viable solution to the detection problem must eventually involve the use of timed models and quantitative analysis of these models. Obviously, the timing of event occurrences is critical to any working fault detection scheme (i.e. a fault occurs if some sensor does not output an expected value within some predefined time frame). Of course, including timing requires the designer to have a-priori knowledge of the timing characteristics of the system, and this adds significantly to the work required to specify the system.

There is also work to be done to improve the results of chapter 6. As stated, our initial approach can greatly increase the number of transitions in the taskblock that detects faults ($TBF(do_x)$). To resolve this we can either: apply well known reduction techniques to reduce the number of transitions (i.e. Karnaugh mapping or a similar technique); or we can modify the firing mechanism for the taskblock by considering fault transitions only after expected transitions are checked. In the second method, we envision then that each place in the $TBF(do_x)$ will have at most one transition leading to the fault state. This needs to be investigated further.

To complete the work of chapter 8, we would like to show how to convert an LTL specification into a condition sequence, and a Kripke structure into a condition

system. In this dissertation we showed the other direction, or that we could convert any trim C-Sequence into a LTL formulation, and any finite condition system into a Kripke structure.

Another interesting area of future research would be in the real-time modification of system behavior based on detected and diagnosed faults (fault correction). Fault correction will require further research in the areas of modeling, control code generation, and timing. This problem is obviously complicated by the fact that many faults on many systems are not correctable. In fact, fault correction may only be applicable to systems with a great deal of flexibility in terms of desired behavior or to systems with built-in redundancy.

In this dissertation, we have established the theoretical background for fault detection and diagnosis under partial observation using untimed condition systems. We then applied these notions to: determine a best-case solution; to construct a supervisor that detects faults; and to present a method of rapidly diagnosing faults by exploiting the causal structure of our condition system models. We also showed how to convert a condition sequence into a linear-time temporal logic formula. We have only considered the single-fault case and this is an area for improvement. A notable point of this research is that we do not require the explicit modeling of faults. Our method instead relies on the re-use of models that have already been created for controller synthesis and for creation of state observers.

Copyright © Jeffrey Ashley 2004

Appendix A

Listings of Diagnosis Code in Visual C++

THE TOP LEVEL PROGRAM CODE(.CPP) FOLLOWS:

```
// diagnosis1.cpp


/*****************************************************
   Author:  jeff ashley, University of Kentucky
   Origination Date:  december 2002

*****************************************************/

#include "stdafx.h"
#include <direct.h>
#include <stdio.h>
#include <stdlib.h>
#include "diagnosis1.h"
#include "CommonNames.h"

#include "NetBlockView.h"

#define THIS_PROGRAM_NAME "diagnoser.exe"

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif
#define MAP_FLAG_ALL 1
#define MAP_FLAG_OUTPUT 0


int iMapFlag = MAP_FLAG_OUTPUT;

typedef CList<CString,LPCSTR> StringListType;
CTypedPtrMap<CMapStringToOb, CString, StringListType*> FileCondMap;
```

```
using namespace std;

int _tmain(int argc, TCHAR* argv[], TCHAR* envp[])
{
   int nRetCode = 0;
   char mystring[200];

   CString sArgv2 = "", sArgv1 = "";


   // Standard output indicating which program is running.


   if(argc > 2) {
      // if an optional argument is given, it should be a directory in
      // which to operate.
      if( _chdir( argv[1] )   ) {
         cout<<"diagnosis: ERROR:  Unable to locate the directory:
"<<argv[1]<<"\n";          cout<<"Press any key to leave diagnoser";
         cin>>mystring;
         exit(0);
   }
      sArgv2 = argv[2];
      sArgv2.TrimLeft();
      sArgv2.TrimRight();

      if (sArgv2 != "-all")
      {
         cout<<"diagnosis: ERROR:  Invalid Flag:"<<argv[2]<<"\n";
         cout<<"Proper Usage:  diagnoser.exe  <PATH_TO_.NET_FILES>  \n";

         system("pause");
         exit(0);
      }
      else
      {
         iMapFlag = MAP_FLAG_ALL;
      }
   }
   else if (argc > 1)
   {
      sArgv1 = argv[1];
      sArgv1.TrimLeft();
      sArgv1.TrimRight();
      if(sArgv1 == "-h")
      {
         cout<<"Proper Usage:  diagnoser.exe  <PATH_TO_.NET_FILES>  \n";
```

```
            system("pause");

            exit(0);
         }
         else if( _chdir( argv[1] )   )
         {
            cout<<"diagnoser: ERROR:  Unable to locate the directory:
"<<argv[1]<<"\n";
            system("pause");
            exit(0);
         }
         else
         {
            iMapFlag = MAP_FLAG_OUTPUT;
         }
      }
      else
      {
            cout<<"Proper Usage:  diagnoser.exe  <PATH_TO_.NET_FILES>  \n";


            exit(0);
      }


      // initialize MFC and print and error on failure
      if (!AfxWinInit(::GetModuleHandle(NULL), NULL, ::GetCommandLine(),
0))   {
         // TODO: change error code to suit your needs
         cerr << _T("Fatal Error: MFC initialization failed") << endl;
         nRetCode = 1;
      }
      else
      {
         //main loop

         SystemBlockView theSystem;

         //   THESE FUNCTIONS ARE PRETTY MUCH TO BE USED IN THIS   ORDER

         theSystem.InitializeBlockViews();

         system("cls");

         theSystem.BuildOutNets();

         theSystem.BuildInNets();
```

```
      theSystem.BuildCausalInfluences();

      theSystem.BuildOutputConditions();



      CStringArray cBad[500]; // this is csets bad;
      int numCbad = 0, numInSet;    CString condName;
      char condStr[100];

      CStringArray  unionSets[500], intersectionSet;    // for each cbad
we will have a union set
      while(numCbad !=-1)
      {
         system("cls");

         theSystem.PrintOutputConditions();
         cout << " Diagnosis program - enter Number of CBad sets in
CSetsBad(-1 to exit): ";
         cin >> numCbad;

         // loop to enter faulty conditions
         for(int i=0 ; i < numCbad; i++)
         {   cBad[i].RemoveAll();  // empty this array before filling it
            cout << "          enter Number of conditions in  Cbad set " << i
 << " : ";
            cin >> numInSet;
            for(int j=0;j< numInSet; j++)
            {
               cout <<"enter condition : ";
               cin >>  condStr;
               condName  = (LPCSTR) condStr;
               cBad[i].Add(condName);
            }

         }

         intersectionSet.RemoveAll();
         theSystem.FindCausalInfluences(cBad, numCbad , intersectionSet);
            // print out the Answer
         cout <<endl << "** A diagnosis is as follows: " << endl;

         for( int l=0;l< intersectionSet.GetSize(); l++)
         {
            cout << (LPCSTR) intersectionSet.GetAt(l) << "  " ;

         }
         cout << endl << endl;
         system("pause");
```

```
        }

    }

    return nRetCode;
}
```

THE CODE(.H AND .CPP) FOR THE DIAGNOSIS AND NETBLOCKVIEW CLASSES FOLLOWS:

```
#ifndef __NETBLOCKVIEW__
#define __NETBLOCKVIEW__

/*****************************************************
   Author:  jeff ashley, University of Kentucky
   Origination Date:  december 2002
   NetBlockView.h
   - classes that build block descriptions of the petri net
model.*****************************************************
**/#include "NetStructureClass.h"
#include "CommonNames.h"

using namespace std;


// The data structure-
class NetBlockView
{
public:
   NetBlockView();
   ~NetBlockView();

   bool isAnOutNet;  // does it output from system
   bool isAnInNet;   // is it an input net
   CString netName;

   CStringArray influenceBlocks;
   //  systems that influence this system - built last

   NetBlockView  *next;        // next item in linked list

   CStringArray outConditions;   // a list of outConditions(as Cstrings!!)

   CNetStructureClass blockNet;
      // 1place and 1transition arc from t to p - captures
      // block view of source net

   CNetStructureClass *SourceNet;
      // original net kept so condition names are easily
      // accessible.
};
```

```
class SystemBlockView
{
public:

   SystemBlockView();
   ~SystemBlockView();
   bool InitializeBlockViews();
         //USE FIRST initializes basic block net.  no in out
            //flags set, no caussal influences built
   void PrintBlocks();
            // basic print of net blocks – i think you can use it
            // right after initializing

      void PrintCausalInfluences(); // use fourth

      bool BuildOutNets();  // USE SECOND FOR BEST RESULTS

   void PrintOutNets();

   void BuildOutputConditions(); // USE LAST builds
            // outputConditions for each block these are stored in
            // a CStringArray

      bool BuildInNets();  // use third but may not be needed

   void PrintInNets();

      void PrintOutputConditions();  // print output Cs for system

   bool BuildCausalInfluences();
         // for each net,  this function builds a CString array
         //of net names representing the influencing subsystems

   void FindCausalInfluences(CStringArray [] ,int numCbad,
CStringArray & intersectionSet);
         // returns the answer to the diagnosis from ACC
         //2002, etc. pass in an array of CStringArrays
         //representing the cbad sets in csets bad and
         // the answer is placed in interSectionSet.

private:
      NetBlockView     *TheSystem;

};

#endif
```

```
/*****************************************************
   Author:  jeff ashley, University of Kentucky
   Origination Date:  december 2002
   NetBlockView.cpp
*****************************************************/

#include "stdafx.h"
#include "NetBlockView.h"



NetBlockView::NetBlockView()
{

   isAnOutNet = false;
   isAnInNet = false;

}


NetBlockView::~NetBlockView()
{
   SourceNet->DeleteAllItems();
   delete SourceNet;

}


SystemBlockView::SystemBlockView()
{

   TheSystem = NULL;

}

SystemBlockView::~SystemBlockView()
{


}

bool SystemBlockView::BuildOutNets()
{

   NetBlockView * currBlock, *blockToCompare;
   CConditionSetClass *currSet, *compareSet;
   CPlaceIdentifierClass *psys ;
   CTransitionIdentifierClass *ptran;
```

```
   bool isAOutNet ;
   POSITION pos1, pos2;
   CConditionIdentifierClass * cid, *cid2;



   currBlock = TheSystem;

   //main loop block to check others against
   while(currBlock !=NULL)
   {
      psys = currBlock->blockNet.GetPlaceID("p1");

      currSet = currBlock->blockNet.GetConditionSetForPlace(psys);
      // get currblock output conditions
      blockToCompare = TheSystem;
      isAOutNet = true;

      while(blockToCompare != NULL  && isAOutNet)
      // inner loop to cycle through blocks to compare
      {

         if((currBlock->netName) != blockToCompare->netName )
         {


            ptran = blockToCompare->blockNet.GetTransitionID("t1");
            compareSet =
blockToCompare->blockNet.GetConditionSetForTransition(ptran);
// get blokc to compare input conditions
            if(currSet)   // adds output conditions to new place
            {
               pos1 = currSet->GetHeadPosition();

               while(pos1!=NULL)
               {
                     cid = currSet->GetNext(pos1);


                  if(compareSet)   // adds output conditions to new place
                  {
                     pos2 = compareSet->GetHeadPosition();

                     while(pos2!=NULL)
                     {
                        cid2 = compareSet->GetNext(pos2);
                        if(currBlock->SourceNet->GetConditionName(cid) ==
blockToCompare->SourceNet->GetConditionName(cid2))
                        {
```

```
                        isAOutNet = false;
                    }
                }
                //cout << endl;

            }
        }
    }

    blockToCompare = blockToCompare->next;

}

// change boolean flag for current block to signify it is an outnet
if(isAOutNet)
{
    currBlock->isAnOutNet = true;
}

currBlock = currBlock->next;

}

return 0;
}

void SystemBlockView::BuildOutputConditions()
{
    NetBlockView *block;
    CPlaceIdentifierClass *psys;
    POSITION pos1;
    CConditionSetClass *cset;


    CString conditionname, conditionname2;
    block = TheSystem;
    CConditionIdentifierClass *cid;
    while(block !=NULL)
    {
        psys =   block->blockNet.GetPlaceID("p1");
        cset =   block->blockNet.GetConditionSetForPlace(psys);
        if(cset)
        {

            pos1 = cset->GetHeadPosition();

            while(pos1!=NULL)
            {
```

```
                cid = cset->GetNext(pos1);
                conditionname = block->SourceNet->GetConditionName(cid);

                block->outConditions.Add(conditionname);


            }
        }

    block = block->next;
    }


}


bool SystemBlockView::BuildInNets()
{

    NetBlockView * currBlock, *blockToCompare;
    CConditionSetClass *currSet, *compareSet;
    CPlaceIdentifierClass *psys ;
    CTransitionIdentifierClass *ptran;
    bool isAInNet ;
    POSITION pos1, pos2;
    CConditionIdentifierClass * cid, *cid2;



    currBlock = TheSystem;

    //main loop block to check others against
    while(currBlock !=NULL)
    {
        ptran = currBlock->blockNet.GetTransitionID("t1");

        currSet = currBlock->blockNet.GetConditionSetForTransition(ptran);
// get currblock output conditions
        blockToCompare = TheSystem;
        isAInNet = true;

        while(blockToCompare != NULL  && isAInNet)
        // inner loop to cycle through blocks to compare
        {

            if((currBlock->netName) != blockToCompare->netName )
            {
```

```
            psys = blockToCompare->blockNet.GetPlaceID("p1");
            compareSet =
blockToCompare->blockNet.GetConditionSetForPlace(psys);
// get blokc to compare input conditions
            if(currSet)   // adds output conditions to new place
            {
               pos1 = currSet->GetHeadPosition();

               while(pos1!=NULL)
               {
                     cid = currSet->GetNext(pos1);

                  if(compareSet)   // adds output conditions to new place
                  {
                     pos2 = compareSet->GetHeadPosition();

                     while(pos2!=NULL)
                     {
                        cid2 = compareSet->GetNext(pos2);
                        if(currBlock->SourceNet->GetConditionName(cid) ==
blockToCompare->SourceNet->GetConditionName(cid2))                    {
                           isAInNet = false;
                        }
                     }
                  }
               }
            }

            blockToCompare = blockToCompare->next;
         }

// change boolean flag for current block to signify it is an outnet
         if(isAInNet)
         {
            currBlock->isAnInNet = true;
         }

         currBlock = currBlock->next;

      }

return 0;
}
```

```
bool SystemBlockView::BuildCausalInfluences()
{

   NetBlockView *block, *inflBlockPtr, *blockToCheck;
   CPlaceIdentifierClass *psys;
   CTransitionIdentifierClass *ptran;
   CConditionSetClass *cset, *cset2;
   CConditionIdentifierClass *cid, *cid2;
   POSITION pos1,pos2;
   CString influence1,influence2,conditionname,conditionname2;
   bool found,found2,done;

   block = TheSystem;
// POSITION pos;

   while(block !=NULL)  // outer loop
   {
      if(block->isAnOutNet) // only check out nets
      {

         block->influenceBlocks.Add(    block->netName);

         for(int i = 0; i < block->influenceBlocks.GetSize();i++)
         {
            influence1 = block->influenceBlocks.GetAt(i);


            inflBlockPtr = TheSystem;
            found = false;
            while(inflBlockPtr!=NULL && found==false)
            {                    // stored net name
               if(inflBlockPtr->netName == influence1)
               {
                  found =true;

               }
               else
                  inflBlockPtr = inflBlockPtr->next;
```

```
            }
// we have now isolated the net bolongtin to the cstring from
//influence blocks.
            blockToCheck = TheSystem;
            while(blockToCheck !=NULL)
            {

               found2 = false;
               for(int j = 0; j < block->influenceBlocks.GetSize();j++)
               {
                  influence2 = block->influenceBlocks.GetAt(j);
                  if(blockToCheck->netName == influence2)
                     found2 = true;

               }

               if(!found2)
               // if not in the influenceBlocks then check to see
               //if  blockToCheck  influences inflBlockPtr
               {
                  psys = blockToCheck->blockNet.GetPlaceID("p1");
                  cset =
blockToCheck->blockNet.GetConditionSetForPlace(psys);
                  if(cset)
                  {
                     pos1 = cset->GetHeadPosition();
                     while(pos1!=NULL)
                     {
                        cid = cset->GetNext(pos1);
                        conditionname =
blockToCheck->SourceNet->GetConditionName(cid);
// now we need to check condition name
//to each input condition to inflBlockPtr
// if so add blockToCheck to influenceBlocks
                        ptran = inflBlockPtr->blockNet.GetTransitionID("t1");
                        cset2 =
inflBlockPtr->blockNet.GetConditionSetForTransition(ptran);

                        if(cset2)
                        {
                           pos2 = cset2->GetHeadPosition();
                           done = false;
                           while(pos2!=NULL  && !done)
                           {
                              cid2 = cset2->GetNext(pos2);
                              conditionname2 =
inflBlockPtr->SourceNet->GetConditionName(cid2);
                              if(conditionname == conditionname2)
                              // hey this block inputs to a block
```

```
                                {
                                    done = true;
                                    block->influenceBlocks.Add(blockToCheck->netName);

                                }
                            }
                        }

                    }

                }
            }

            blockToCheck= blockToCheck ->next;
        }
    }
}

    block = block->next;
}


Return 0;
}



void SystemBlockView::FindCausalInfluences(CStringArray cBad[] ,int
numCbad, CStringArray & intersectionSet)
{    // intersectionSet is the answer to the diagnosis question




    CString condName;

    NetBlockView *block;
    CStringArray unionSets[500]; // intersectionSet;    // for each cbad
we will have a union set


    bool found;

    intersectionSet.RemoveAll();

    // build union Sets - whew!
    for(int k=0 ; k < numCbad; k++) // union Set and cBad index!!!!!
    { unionSets[k].RemoveAll();
      for(int l=0;l< cBad[k].GetSize(); l++)
```

```
        {
            block = TheSystem;
            found = false;
            while(block !=NULL && !found)
            // find block outputting conditionGetAt(l)
            {
               for(int p=0;p< block->outConditions.GetSize(); p++)
               {
                  if(block->outConditions.GetAt(p) == cBad[k].GetAt(l))
                  {
                     found = true;
                  }


                  if(found)  // found the block
                  {
                     // add netNames to union set
                     for(int  r=0; r< block->influenceBlocks.GetSize(); r++)
                     {

                        bool found2;
                        found2 = false;
                        for(int q=0; q< unionSets[k].GetSize(); q++)
            // to make sure influence block isn't in union yet
                        {
                           if (unionSets[k].GetAt(q)
== block->influenceBlocks.GetAt(r))
                           {
                           // already in unionSets
                              found2 = true;
                           }
                        }
                        if(!found2)
                        {
                           // add  influence block to unionSets[k];
                           unionSets[k].Add(block->influenceBlocks.GetAt(r));
                        }
                     }

                  }

               }
               block = block->next;
            }

        }


    }
```

```
        bool found3, found4;

        //  determine intersection set
        if(numCbad == 1)  // one CBAD so no intersection
        {
           intersectionSet.Copy(unionSets[0]);

        }

        else  // more than one union set
        {
           for(int i = 0; i< unionSets[0].GetSize();i++)
           {
              found3=true;

              for(  k=1 ; k < numCbad; k++)
              {
                 found4=false;
                 for( int l=0;l< unionSets[k].GetSize(); l++)
                 {
                    if(unionSets[0].GetAt(i) == unionSets[k].GetAt(l))
                       found4=true;

                 }
                 found3 = found3 && found4;
              }
              if(found3)
                 intersectionSet.Add(unionSets[0].GetAt(i));

           }
        }

}



void SystemBlockView::PrintOutNets()
{
   NetBlockView *block;


   block = TheSystem;
  cout << " Output Nets are " << endl;

   while(block !=NULL)
   {
      if(block->isAnOutNet)
```

```
        cout << (LPCSTR)block->netName<< endl;
        block = block->next;

    }


}



void SystemBlockView::PrintInNets()
{
    NetBlockView *block;


    block = TheSystem;
   cout <<endl<< " Input Nets are " << endl;

    while(block !=NULL)
    {
        if(block->isAnInNet)
        cout << (LPCSTR)block->netName<< endl;
        block = block->next;

    }







}



void SystemBlockView::PrintBlocks()
{
    CPlaceIdentifierClass *psys;
    CTransitionIdentifierClass *ptran;
    CConditionSetClass *cset;
    CConditionIdentifierClass *cid;
    POSITION pos1;
    NetBlockView *ptr;
    CString conditionname;

    ptr = TheSystem;
```

```
    while(ptr  != NULL)
    {
    cout<<endl <<"NET NAME -> " << (LPCSTR) ptr->netName <<  endl;
       psys =   ptr->blockNet.GetPlaceID("p1");
       cset =   ptr->blockNet.GetConditionSetForPlace(psys);

    // cout <<(LPCSTR)ptr->blockNet.GetPlaceName(psys) << endl;

       cout << endl << " Output Conditions " << endl;

       if(cset)
       {
          //cout << "in if";
          pos1 = cset->GetHeadPosition();



          while(pos1!=NULL)
          {


                cid = cset->GetNext(pos1);

//           conditionname = ptr->blockNet.GetConditionName(cid);

             conditionname = ptr->SourceNet->GetConditionName(cid);
             if (conditionname.GetLength() > 0)
             {

                cout << "   " << (LPCSTR)conditionname << endl;


             }
             else
             {
                cout << "can't find name for condition\n";
             }



          }


       }


       cout << endl << " Input Conditions " << endl;

       ptran =  ptr->blockNet.GetTransitionID("t1");
```

```
        cset =   ptr->blockNet.GetConditionSetForTransition(ptran);


        if(cset)
        {
        // cout << "in if";
           pos1 = cset->GetHeadPosition();
           while(pos1!=NULL)
           {


                 cid = cset->GetNext(pos1);

//             conditionname = ptr->blockNet.GetConditionName(cid);

               conditionname = ptr->SourceNet->GetConditionName(cid);
               if (conditionname.GetLength() > 0)
               {

                  cout << "  " << (LPCSTR)conditionname << endl;
        //        cout << "in while";

               }
               else
               {
                  cout << "can't find name for condition\n";
               }



           }


        }




        ptr = ptr->next;


     }


}


void SystemBlockView::PrintCausalInfluences()
{
```

```
    NetBlockView *block;

    CString influence;
    block = TheSystem;

    while(block != NULL)
    {
        if(block->isAnOutNet)
        {
            cout << endl << (LPCSTR)block->netName << " is influenced by " <<
endl;
            for(int i = 0; i < block->influenceBlocks.GetSize();i++)
                cout << (LPCSTR) block->influenceBlocks.GetAt(i) << endl;

        }

        block = block->next;
    }

}

bool SystemBlockView::InitializeBlockViews(){


    // look for all net files and process each one.
    //
    CNetStructureClass *SourceNet;

    CConditionSetClass *cset ;

    CConditionIdentifierClass *cid;


    CString netfilename,conditionname,  transitionname, placename;



    CFileFind finder;
    CTime netfiletime;
    BOOL bFound;
    POSITION    pos1;
    NetBlockView  *block;

    CPlaceIdentifierClass *ppl;
    CTransitionIdentifierClass *ptr;

    int updatecount = 0;
```

```
   // Now we read in the net files and look at their time stamps.
   bFound = finder.FindFile(NET_FILE_PATTERN);
   if(!bFound) {
      cerr<<endl<<"No net files found in the directory"<<endl;
      return 1;
   }


   cout << "NETS LOADED ARE" << endl;
   while (bFound)
   {


//    cout << "IN BFOUND " << endl;



      bFound = finder.FindNextFile();
      //netfilename = finder.GetFilePath();
      netfilename = finder.GetFileName();



////////////////////////////////////////////////////////////////
//read in FromNet from file.        SourceNet = new CNetStructureClass;
      if (!SourceNet->ReadNetFromFile(netfilename) )
      {
//    if(! FromNet.ReadNetFromFile(netfilename) )
//    {
         cerr << "ERROR: Unable to open requested input file:
"<<(LPCTSTR)netfilename <<endl;        exit( 1 );
      }

//       cset = FromNet.GetSetOfAllConditions();
// JEFF

      cout << (LPCTSTR)netfilename << endl;

      block = new NetBlockView;

      block->netName = netfilename;
      block->next = TheSystem;
      block->SourceNet = SourceNet;

      TheSystem = block;
      ppl = block->blockNet.NewPlace("p1");
      ptr = block->blockNet.NewTransition("t1");
      block->blockNet.AddArcToFrom(ptr,ppl);
      cset = SourceNet->GetSetOfAllConditions();
```

```
// block->InConditions.GetHeadPosition();
      if(cset)
      {
         pos1 = cset->GetHeadPosition();
         while(pos1!=NULL)
         {
            cid = cset->GetNext(pos1);
            conditionname = SourceNet->GetConditionName(cid);
            if( SourceNet->GetTransitionSetForCondition(cid) )
            {
               cout << "input condition - " << (LPCTSTR) conditionname <<endl;
               block->blockNet.AddTransitionCondition(ptr,cid);

            }
            else if( SourceNet->GetPlaceSetForCondition(cid) )
            {


               block->blockNet.AddPlaceCondition(ppl,cid);
               cout << "output condition - " << (LPCTSTR)conditionname << endl;
            }
            //else
            // cout  <<"neither " << (LPCTSTR)conditionname << endl;
         }

      }

   }


// FromNet.DeleteAllItems();


   return 0;


}


void SystemBlockView::PrintOutputConditions()
{
   NetBlockView *block;
// CPlaceIdentifierClass *psys;
// POSITION pos1;
// CConditionSetClass *cset;
   cout << " OUTPUT CONDITIONS OF THE SYSTEM ARE " << endl;
   CString conditionname;
   block = TheSystem;
   //CConditionIdentifierClass *cid;
```

```
    while(block !=NULL)
    {
        if(block->isAnOutNet)
        {

            for(int i = 0; i < block->outConditions.GetSize();i++)

                {conditionname= block->outConditions.GetAt(i);
            //     cid = cset->GetNext(pos1);
            //     conditionname = block->SourceNet->GetConditionName(cid);
                // cout << "    " << (LPCSTR) block->outConditions.GetAt(i)<<endl;
                    cout << "   " << (LPCSTR)conditionname << endl;
                }
//          }
        }
    block = block->next;
    }
}
```

Bibliography

[AFB+98] Armen Aghasaryan, Eric Fabre, Albert Benveniste, Renee Boubour, and Claude Jard. Fault detection and diagnosis in distributed systems: an approach by partially stochastic petri nets. DISCRETE EVENT DYNAMIC SYSTEMS: THEORY AND APPLICATIONS ; KLUWER ACADEMIC PUBLISHERS, 8(1):203–231, 1998.

[AH02a] Jeffrey Ashley and Lawrence E. Holloway. Qualitative diagnosis of condition systems. In PROCEEDINGS OF THE AMERICAN CONTROLS CONFERENCE CONFERENCE, Anchorage, Alaska, May 2002.

[AH02b] Jeffrey Ashley and Lawrence E. Holloway. Qualitative diagnosis of condition systems. SUBMITTED TO: DISCRETE EVENT DYNAMIC SYSTEMS: THEORY AND APPLICATIONS ; KLUWER ACADEMIC PUBLISHERS, 2002.

[CL01] C. Cassandras and S. Lafortune. INTRODUCTION TO DISCRETE EVENT SYSTEMS. Kluwer Academic Publishers, 2001.

[Dav84] R. Davis. Diagnostic reasoning based on structure and behavior. ARTIFICIAL INTELLIGENCE, 24(3):347–410, December 1984.

[deK86a] Johan deKleer. An assumption based tms. ARTIFICIAL INTELLIGENCE, 28(2), 1986.

[deK86b] Johan deKleer. Extending the atms. ARTIFICIAL INTELLIGENCE, 28(2), 1986.

[DH88] Randall Davis and Walter C. Hamscher. Model-based reasoning: Troubleshooting. Technical Report 1059, Massachussets Institute of Technology Artificial Intelligence Laboratory, July 1988.

[DP94a] Adnan Darwiche and Judea Pearl. Symbolic causal networks. In PROCEEDINGS OF THE 12TH NATIONAL CONFERENCE ON ARTIFICIAL INTELLIGENCE(AAAI), pages 238–244, 1994.

[DP94b] Adnan Darwiche and Judea Pearl. Symbolic causal networks for reasoning about actions and plans. Technical report, Working Notes:AAAI Spring Symposium on Decision Theoretic Planning, 1994.

[dW87] Johan deKleer and B.C. Williams. Diagnosing multiple faults. ARTIFICIAL INTELLIGENCE, 32(1), 1987.

[GH00] Yu Gong and L. E. Holloway. State observer synthesis for a class of condition systems. In PROCEEDINGS OF THE 5TH WORKSHOP ON DISCRETE EVENT SYSTEMS(WODES2000). Kluwer Academic Publishers, 2000.

[GH01] Yu Gong and L. E. Holloway. Multi-layer state observers for condition systems. In PROCEEDINGS OF THE IEEE INTERNATIONAL CONFERENCE ON EMERGING TECHNOLOGIES AND FACTORY AUTOMATION (ETFAÕ01), Antibes, Nice, France, October 2001.

[GO94] D.M. Gabbay and H.J. Ohlbach(eds.). TEMPORAL LOGIC FROM THE LECTURE NOTES IN ARTIFICIAL INTELLIGENCE. Springer-Verlag, 1994.

[HA98a] L. Holloway and J. Ashley. Condition languages and condition systems for modeling ambiguous control specifications. In IEE INTERNATIONAL WORKSHOP ON DISCRETE EVENT SYSTEMS (WODES98), Cagliari, August 1998.

[HA98b] L. E. Holloway and J. Ashley. Elaborative orderings of condition languages. In PROCEEDINGS OF 1998 IEEE CONFERENCE ON DECISION AND CONTROL, Tampa, December 1998.

[Had03] Christoforos N. Hadjicostis. Nonconcurrent error detection and correction in fault-tolerant linear finite-state machines. IEEE TRANSACTIONS ON AUTOMATIC CONTROL, 48(12):2133–2140, 2003.

[HCJK03] Z. Huang, V. Chandra, S. Jiang, and R. Kumar. Modeling discrete event systems with faults using a rules based formalism. MATHEMATICAL AND COMPUTER MODELING OF DYNAMICAL SYSTEMS, 9(3):233–252, 2003.

[HGSA00] L. E. Holloway, X. Guan, R. Sundaravadivelu, and J. Ashley. Automated synthesis and composition of taskblocks for control or manufacturing systems. IEEE TRANSACTIONS ON SYSTEMS, MAN AND CYBERNETICS: PART B, 30(5):696–712, October 2000.

[HLR97] H.-M. Hanisch, A. Luder, and M. Rausch. Controller synthesis for net condition/event systems with a solution for incomplete state observation. EUROPEAN JOURNAL OF CONTROL, (3):280–291, 1997.

[HR00] M. Huth and M. Ryan. LOGIC IN COMPUTER SCIENCE: MODELLING AND REASONING ABOUT SYSTEMS. Cambridge University Press, 2000.

[JdKR92] Alan Mackworth Johan de Kleer and Raymond Reiter. Characterizing diagnoses and systems. ARTIFICIAL INTELLIGENCE, 56:197–222, 1992.

[JHCK01] Shengbing Jiang, Zhongdong Huang, Vigyan Chandra, and Ratnesh Kumar. A polynomial algorithm for testing diagnosability of discrete event systems. IEEE TRANSACTIONS ON AUTOMATIC CONTROL, 46(8):1318–1321, August 2001.

[JK03a] S. Jiang and R. Kumar. Diagnosis of repeated failures for discrete event systems with linear-time temporal logic specifications. SUBMITTED TO: IEEE TRANSACTIONS ON SYSTEMS, MAN, AND CYBERNETICS, PART B, 2003.

[JK03b] S. Jiang and R. Kumar. Failure diagnosis of discrete event systems with linear-time temporal logic fault specifications. SUBMITTED TO: IEEE TRANSACTIONS ON AUTOMATIC CONTROL, 2003.

[JK03c] S. Jiang and R. Kumar. Supervisory control of discrete event systems with ctl* temporal logic specifications. SUBMITTED TO: SIAM JOURNAL ON CONTROL AND OPTIMIZATION, 2003.

[JKG03] Shengbing Jiang, Ratnesh Kumar, and Humberto E. Garcia. Diagnosis of repeated failures in discrete event systems. IEEE TRANSACTIONS ON ROBOTICS AND AUTOMATION, 19(2):310–323, April 2003.

[KLLU98] S. Kowalewski, Y. Lakhnech, B. Lukoschus, and L. Urbina. On the composition of condition/event systems. In PROCEEDINGS OF INTERNATIONAL WORKSHOP ON DISCRETE EVENT SYSTEMS, WODES98, Cagliari, August 1998. IEE.

[Kro87] Fred Kroger. TEMPORAL LOGIC OF PROGRAMS. Springer-Verlag, 1987.

[MP91] Z. Manna and A. Pnuelli. THE TEMPORAL LOGIC OF REACTIVE AND CONCURRENT SYSTEMS. Springer-Verlag, 1991.

[O'C91] P. O'Conner. PRACTICAL RELIABILITY ENGINEERING. John Wiley & Sons, 1991.

[PC98] Gregory Provan and Yi-Liang Chen. Diagnosis of timed discrete event systems using temporal causal networks: Modeling and analysis. In PROCEEDINGS OF THE 4TH WORKSHOP ON DISCRETE EVENT SYSTEMS(IEE), pages 152–154, 1998.

[PC00] Gregory Provan and Yi-Liang Chen. Characterizing controllability and observability properties of temporal causal network modeling for discrete event systems. In PROCEEDINGS OF THE AMERICAN CONTROLS CONFERENCE, pages 3540–3544, 2000.

[SD89] Peter Struss and Oskar Dressler. Physical negation: Integrating fault models into the general diagnostic engine. In PROCEEDINGS OF THE 11TH INTERNATIONAL JOINT CONFERENCE OF ARTIFICIAL INTELLIGENCE(IJCAI-89), volume 2, pages 1318–1323, 1989.

[SK91] R.S. Sreenivas and B.H. Krogh. On condition/event systems with discrete state realizations. DISCRETE EVENT DYNAMIC THEORY AND APPLICATIONS, 1(2), September 1991.

[SSL+94] M. Sampath, R. Sungupta, S. Lafortune, K. Sinnamohideen, and D. Teneketzis. Diagnosability of discrete event systems. In PROCEEDINGS OF 11TH INTERNATIONAL CONFERENCE ON ANALYSIS AND OPTIMIZATION OF SYSTEMS: DISCRETE EVENT SYSTEMS, Sophia-Antipolis, France, June 1994.

[SSL+95] Meera Sampath, Raja Sengupta, Stephane Lafortune, Kasim Sinnamohideen, and Demosthenis C. Teneketzis. Diagnosability of discrete event systems. IEEE TRANSACTIONS ON AUTOMATIC CONTROL, 40(9):1555–1575, 1995.

[SSL+96] Meera Sampath, Raja Sengupta, Stephane Lafortune, Kasim Sinnamohideen, and Demosthenis C. Teneketzis. Failure diagnosis using discrete-event models. IEEE TRANSACTIONS ON CONTROL SYSTEMS TECHNOLOGY, 4(2):105–124, 1996.

[ZKW99] S. Hashtrudi Zad, R. H. Kwong, and W.M. Wonham. Fault diagnosis in timed discrete-event systems. In PROCEEDINGS OF THE 38TH CONFERENCE ON DECISION AND CONTROL, pages 1756–1761, 1999.

[ZKW03] S. Hashtrudi Zad, R. H. Kwong, and W.M. Wonham. Fault diagnosis in discrete-event systems: Framework and model reduction. IEEE TRANSACTIONS ON AUTOMATIC CONTROL, 48(7), July 2003.

[ZM99] S. Hashtrudi Zad and M. Massoumnia. Generic solvability of the failure detection and identification problem. AUTOMATICA, 35:887–893, 1999.

Vita

Date and place of birth

January 4, 1966 in Lexington, KY.

Educational institutions attended and degrees awarded

M.S.M.S.E., University of Kentucky, Lexington, Kentucky.
Specialized in Discrete Event Systems applications to Manufacturing.
Thesis: Colored Controlled Petri Nets.
B.S.E.E., University of Kentucky, Lexington, Kentucky.

Professional positions held

1. 2001-present.

   Assistant Professor, Division of Computer and Technical Sciences, Kentucky State University, Frankfort, KY.

2. 1999-2001.

   Instructor, Electronics program, Spencerian College, Lexington, KY.

3. 1998-1999.

   Engineer, Federal Signal Corporation, Danville, KY.

4. 1993-1998.

   Research Assistant, College of Engineering, University of Kentucky, Lexington, KY.

5. 1990-1992.

   Electronic Reliability Engineer, Whirlpool Corp., Evansville, IN.

6. 1987-1989.

   Engineering Co-op, General Electric, Louisville, KY.

Scholastic and professional honors

1. 1993 - 1998.

   Awarded a research assistantship by the Department of Electrical Engineering and the Center for Manufacturing Systems, University of Kentucky,

2. 1992.

   Awarded status of Certified Reliability Engineer from the American Society for Quality(ASQ).

3. 1989.

   Member of Eta Kappa Nu.

Journal submissions

1. L.E. Holloway, X. Guan, R. Sundaravadivelu, and J. Ashley Jr. AUTOMATED SYNTHESIS AND COMPOSITION OF TASKBLOCKS FOR CONTROL OF MANUFACTURING SYSTEMS. IEEE Transactions on Systems, Man, and Cybernetics Part B, volume 30(5), October 2000, pp. 696-712.

2. Jeffrey Ashley and L. E. Holloway. QUALITATIVE DIAGNOIS OF CONDITION SYSTEMS. Submitted to The Journal of Discrete Event Dynamic Systems, Kluwer Academic Publishers.

Conference papers

1. Jeffrey Ashley Jr. and L. E. Holloway. CHARACTERIZING UNCONTROLLABLE REACHABILITY FOR COLORED CONTROLLED PETRI NETS. Proceedings of the 1994 IEEE International Conference on Systems, Man, and Cybernetics, San Antonio, Texas, October, 1994. pp. 1000-1005.

2. Jeffrey Ashley Jr. and L. E. Holloway. SUPERVISORY CONTROL OF CONCURRENT TASK SEQUENCES OPERATING OVER LOW-LEVEL DEVICES. 35th Annual Allerton Conference on Communication, Control, and Computing, October, 1997.

3. L. E. Holloway and Jeffrey Ashley Jr. CONDITION LANGUAGES AND CONDITION SYSTEMS FOR MODELING AMBIGUOUS CONTROL SPECIFICATIONS. 1998 Workshop on Discrete Event Systems (WODES98), Cagliari Italy, August 1998.

4. L. E. Holloway and J. Ashley Jr. ELABORATIVE ORDERINGS OF CONDITION LANGUAGES. 1998 IEEE Conference on Decision and Control, Tampa, December 1998.

5. Jeffrey Ashley Jr. and L. E. Holloway. DIAGNOSIS OF CONDITION SYSTEMS USING DIAGNOSTIC CAUSAL NETWORKS . Proceedings of the 2001 IEEE International Conference on Systems, Man, and Cybernetics; Tuscon, Arizona; October, 2001 .

6. Jeffrey Ashley Jr. and L. E. Holloway. DIAGNOSIS OF CONDITION SYSTEMS USING CAUSAL STRUCTURE. Proceedings of the 2002 American Controls Conference; Anchorage, Alaska; May, 2002.