Automated Control, Observation, and Diagnosis of Multi-layer Condition Systems[1]

Jeffrey Ashley

Center for Manufacturing Systems
University of Kentucky, Lexington,
Kentucky 40506

ashley@engr.uky.edu,

Lawrence E. Holloway

Dept. of Electrical and Computer Engineering and
Center for Manufacturing
University of Kentucky, Lexington, Kentucky 40506, USA

holloway@engr.uky.edu

Abstract: A condition system is a distributed Petri Net where model to model communication is done via state-based signals called conditions. In this paper, the application of the authors' research in controller synthesis techniques using condition systems is presented. First, an overall perspective is presented. Next, the focus is on the major components used in this approach. Namely these are: the plant models of the unconstrained open-loop system to be controlled; the synthesized observer models that provide state estimation; the synthesized controller models that provide closed-loop control; and failure detection, diagnosis and reconfiguration. A simple robotic arm is used for an example. The paper concludes with a brief overview of the new version of the software tool developed by the authors for use in this research.

Dr. Jeffrey Ashley is currently a research scientist for the University of Kentucky. He completed his Ph.D. in Electrical and Computer Engineering in 2004 at the University of Kentucky. He received a B.S.E.E. and a Master's in Manufacturing Systems Engineering from the same institution. He has worked in industry for six years in advanced manufacturing engineering, and in product design and reliability engineering. He has also worked in full-time instruction for six years, including three years as an Assistant Professor at Kentucky State University in the field of Computer Science.

Dr. Larry Holloway is the Director of the UK Center for Manufacturing and the Kentucky Utilities Professor of Electrical and Computer Engineering. He received his BS in Electrical Engineering from Southern Methodist University in Dallas Texas, and his MS and PhD in Electrical and Computer Engineering from Carnegie Mellon University in Pittsburgh. Dr Holloway joined the University of Kentucky in 1991 as a joint faculty member in Electrical Engineering and the Center for Manufacturing. His research area is in analysis and control of systems, particularly applied to manufacturing. He has conducted projects funded by the National Science Foundation, Eaton, Rockwell, Department of Defense, NASA, Office of Naval Research, and others. He has over 100 technical publications. He is formerly an Associate Editor of IEEE Transactions on Automatic Control, and has served as a member of the Program Committee or Organizing Committee of ten international conferences.

1. Introduction

The field of Discrete Event Systems *DES*, introduced by [**Error! Reference source not found.**] and [**Error! Reference source not found.**], was motivated by the novel idea of extending traditional systems theory to discrete state systems. Since then, it has been an active topic of research. The principal goal of much of this research (although not exclusively) has been in the *synthesis* of controllers from discrete state models (i.e. to automatically generate controllers from models of the system). Condition systems are a subset of condition/event systems and were first considered by [**Error! Reference source not found.**]. Our research in condition systems began with the presentation of our specification and analysis mechanism called a condition set sequence in [**Error! Reference source not found.**, **Error! Reference source not found.**]. We then considered controller synthesis in [**Error! Reference source not found.**]. We were motivated to consider the use of condition systems, a labeled Petri net, because it is a natural model for modular design and synthesis. We have since extended this research in ways highlighted within this paper.

As a brief overview of our approach, we first create a set of condition system models that describe the unconstrained behavior of some open-loop plant we wish to control. From this set of models and a set of specifications we synthesize most of the models that are needed to implement a closed-loop control of the plant. In this paper we focus on: the nature of these open-loop plant models; the observer models that provide a state estimate of the plant; and the taskblock models that implement closed-loop control. Failure detection, diagnosis and control reconfiguration are also considered. We will also briefly discuss the new version of our control synthesis and analysis software package called *Spectool*.

The Spectool project at the University of Kentucky is a project to develop a set of techniques and software tools for analysis, monitoring, control, and code development for a class of discrete systems. The approach was motivated initially by the problem of developing and debugging of logic control software for industrial systems. Given sufficient knowledge of the industrial system as represented by models, the

1 This work has been supported in part by the National Science Foundation grant ECS-0115694, Office of Naval Research N000140110621, and the Center for Manufacturing at the University of Kentucky.

goal is to use that knowledge to automatically synthesize correct control software according to high-level specifications, and to synthesize monitors for automated fault detection and diagnosis.

The approach demands that the system knowledge must be represented by a modeling framework with the following characteristics:

- **The system model must be composed of reusable subsystem models.** The goal is to reduce the burden of modeling the system by using common *plug-and-play* subsystem models, perhaps from libraries of common devices (actuators, sensors, conveyors, etc.), perhaps eventually supplied directly by the device manufacturer.
- **The subsystem models need to interact through well-defined input and output structures appropriate for modeling of physical devices.** This demands that at the least the modeling framework need a notion of condition signals (signals with a value over time, as opposed to instantaneous events). For example, if a part is on a conveyor, the change of state of the part depends on the current state of the conveyor. Thus, the models need to interact in a way that a model of the part has current knowledge of information about the state of the conveyor.
- **The model interaction mechanism should be causal.** A cause and effect structure is critical in modular synthesis of control and in automated fault diagnosis.

The models that we use are *condition systems*, a class of the condition-event systems developed by [**Error! Reference source not found.**] and considered by [**Error! Reference source not found.**, **Error! Reference source not found.**] and others. The model is a form of Petri net with explicit input and output signals, called *conditions*. For the class of models considered for subsystems, the condition system models are comparable to state machines with guards. The condition model framework allows systems to be modeled as a set of interacting subsystems. The explicit input-output structures of these subsystem models allow them to be combined together in a manner similar to the wiring together of digital circuits. This simplifies system modeling by allowing the reuse of common subsystem models. It also provides the causal structure necessary or our control synthesis and fault diagnosis.

The control synthesis part of the Spectool project has focused on synthesizing control from a high-level description of desired behavior, and to do this through a series of local synthesis operations on subsystem models. More specifically, we view the control synthesis problem as a "navigation" task of determining the right actuation signals to drive the system from its current state through a series of target goal states, and the task of control synthesis is to determine a safe path between these target states. For example, if one of the target goal states is to have a given product completed at the end of the conveyor, the control synthesis task is to determine the sequence of low level actuation signals (and the sensor signals in response) necessary to bring the incoming part to a completed state at the end. Furthermore, this is done through generating a series of local controllers from analysis of individual subsystem models, and these are then connected sequentially and hierarchically to achieve the final control.

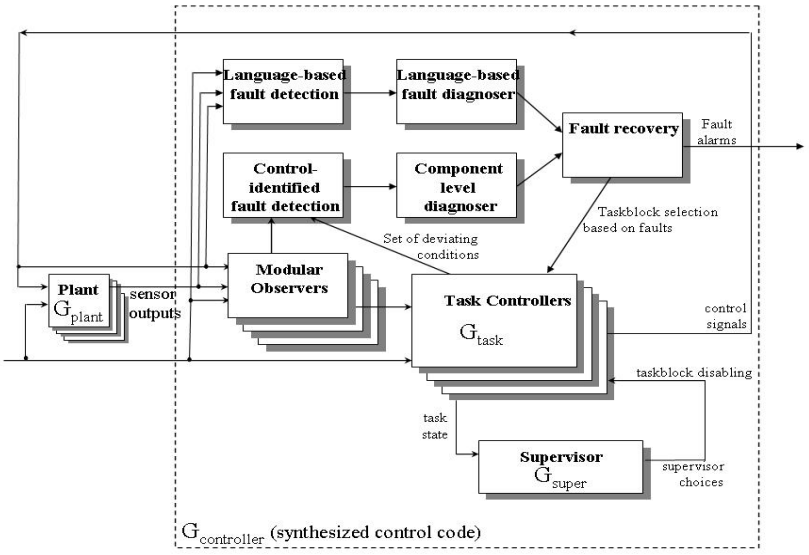

Figure 1: Synthesized Elements of the Control and Monitoring System.

Figure 1 shows the structure of the synthesized systems. Outputs of the interacting plant subsystems are used by the interacting state observer subsystems to determine the system state, which is then used by the interacting control subsystems (called *taskblocks*) to drive the system through desired behaviors. A supervisor is used to coordinate the taskblock subsystems to ensure safety constraints. A fault monitor is used to detect inappropriate system responses (detect faults) and diagnose their cause based on the causal structure of the plant's subsystem interactions. All elements of the controller ($G_{controller}$) have their logic automatically synthesized through analysis of the plant models and given high-level specifications of the desired behavior, and the resulting logic is then automatically converted into C++ objects, which are then compiled into executable software. The synthesis of taskblocks is documented in [**Error! Reference source not found.**]. The synthesis of observers was described in [**Error! Reference source not found.**]. The synthesis of the fault monitors is described in [**Error! Reference source not found.**] [**Error! Reference source not found.**]. The supervisor synthesis is described in [**Error! Reference source not found.**]. The following list describes the elements from the figure in greater detail.

- The *Plant* represents a model of the open-loop behavior for some system we wish to control. It is composed of one or more interacting *subsystem models*, each a condition system, where each represents the open-loop behavior of some subsystem. Within the plant there may be many such subsystem models. The *specification* is a condition set sequence that describes a desired behavior of the system. A specification has an equivalent condition system model. The *specification net* (called the Espec net in [**Error! Reference source not found.**]) is synthesized from the specification. The specification net acts as the top-level model within our control scheme by initiating lower level controllers.

- An *observer* is a synthesized condition system model that is used to estimate the state of a subsystem of the plant. An observer collects dynamic information from the plant to provide this estimate. Thus, a subsystem model state might not be directly evident from observed signals, but as long as the subsystem satisfies certain properties then the observer will be able to provide an accurate estimate of the state. In general, we need an observer for each subsystem model within the plant.

- A *taskblock* is a synthesized condition system model that represents a controller for a specific subsystem model. For some subsystem model there may be many such taskblocks where each represents the controller logic for achievement of a specific task or goal (also representable as a set of conditions).

- We have considered two methods for *fault detection and diagnosis* which is responsible for detecting faulty behaviors and identifying the faulty subsystem. In the first, we have considered an approximate diagnosis scheme using causal networks in [**Error! Reference source not found.**]. We presented a complimentary rapid detection scheme by modifying taskblock synthesis to include fault detection capability in [**Error! Reference source not found.**] and [**Error! Reference source not found.**]. We have also considered a language-based detection and diagnosis approach that provides a "best possible" diagnosis given the constraints of observability in [**Error! Reference source not found.**]. Finally, we have considered *fault reconfiguration* in [**Error! Reference source not found.**].

- *Forbidden state avoidance* handles resource allocation and avoidance of catastrophic states that occur between two or more subsystem models within the plant model. We can also characterize this as *supervisory control* in the spirit of [**Error! Reference source not found.**]. We also use the same mechanism to prevent two taskblocks from targeting contradictory goals. In our initial work in this area, we presented a non-optimal method represented as simple condition system models that can preempt taskblock functioning via a set of conditions that are inputs of potentially affected taskblocks. We will not consider this work within this paper although details can be found in [**Error! Reference source not found.**].

Figure 2 illustrates the synthesis elements and their relation to one another within our scheme. In this diagram, octagons represent specifications and models that must be created by a human, while circles represent synthesized models or control code. Finally, the squares represent implemented software used in this process. The following list briefly discusses each element of this figure.

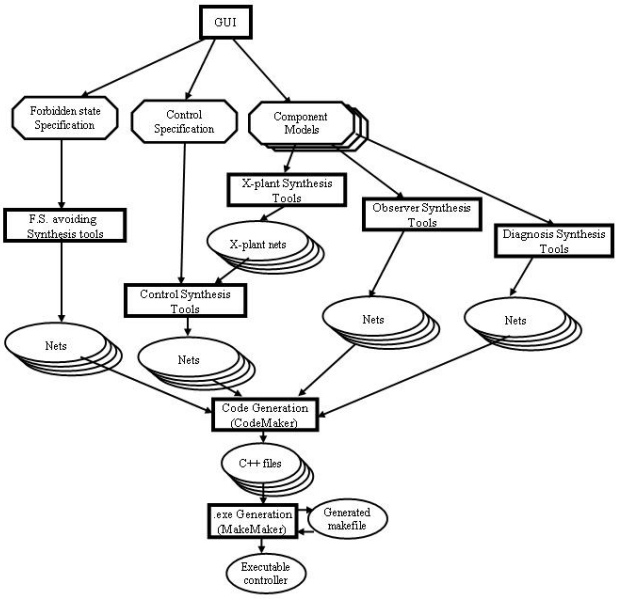

Figure 2: The Synthesis of Controller, Observer, and Other Portions Used within Our Approach

- *GUI* - The Spectool software is used to create the control specification and the subsystem models. Both can be represented as condition systems.

- *Subsystem (or Component) models* - These models describe the unconstrained behavior of the plant elements.

- *Observer synthesis* - These tools synthesize an observer model from the subsystem models.

- *X-plant synthesis tools* - These tools synthesize an intermediate model, the x-plant, for subsystem models in the plant. This model is not used directly for control code generation, but is instead required as an intermediate model for control synthesis. An x-plant represents the merger of the plant model and the observer model.

- *Control synthesis tools* - These tools are responsible for creating models that represent the controller. Our approach allows for the modular synthesis of these models, under certain mild assumptions, and thus each of these models typically encodes a single desired behavior for a single subsystem model.

- *Diagnosis synthesis tools* - Currently we have developed a simple off-line tool for providing approximate diagnosis. Since expanded diagnosis and reconfiguration are subjects of current research no tools are currently available. We also anticipate that fault detection can be achieved (and diagnosis assisted) through a modified controller structure.

- *CodeMaker, MakeMaker, and files* - These are responsible for taking the controller nets and the observer nets and converting them automatically into executable code. We note that to these tools the observer nets and controller nets are indistinguishable. Both are just considered to be condition systems.

This paper is organized as follows. In section 2, we define condition systems and discuss the properties of such systems. We frame the problem of observation for condition systems in section 3. We present taskblocks in section 4, and in section 5 we define the diagnosis problem and discuss areas of possible solution. We also have provided a brief review of our newest version of Spectool. We conclude with a brief discussion.

2. Condition Systems and Plant Models

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 [**Error! Reference source not found.**].

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

The following defines a condition system. We again note that we use this type of model exclusively within this paper. A condition system is 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 [**Error! Reference source not found.**].

Definition 1: A condition system, G is modeled as a labeled Petri Net characterized by a set of places P_G, a set of transitions T_G, a set of directed arcs A_G between places and transitions, and a condition mapping function $\Phi_G(\cdot)$ that maps condition sets to places and transitions. The dynamics of the system are defined in the following manner:

1. *The states are the markings of the Petri net:* The state of the condition system is determined by the markings of the places. A marking for the system is defined as m, and for a single place as $m(p)$. The set of all markings is defined as M_G and the initial marking is defined as m_0 .
2. *The output conditions have their truth value established by marked places:* A condition system outputs conditions according to the marking of the system such that for any place p with $m(p)>0$, then all conditions $c \in \Phi_G(p)$ associated with that place will be true.
3. *Next-state dynamics depend on state enabling and condition enabling:* A transition t can *fire* thereby changing the state of the system if it is state enabled and condition enabled as described below.
   (a) A transition $t \in T$ is *state enabled* if all places p, that are input to t, have a nonzero marking ($m(p) \geq 1$).
   (b) A transition $t \in T$ is *condition-enabled* if all conditions mapped to t via $\Phi_G(t)$ have a truth value of *true*.

   A transition that *fires* removes a token from each place that is input to the transition, and adds a token to each place that in an output of the transition in the usual Petri net manner.

An example of a condition system is shown in figure 3. 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. These subsystems communicate exclusively through conditions where the output conditions of one subsystem may enable and fire a transition in another subsystem. The key advantage to this modeling structure is that it allows for modular design and analysis. For the remainder of this paper we will use the notation G to indicate the complete system, and the notation $\{G_1,...,G_n\}$ to indicate the set of subsystems in G. We also make the following assumption on the structure of our subsystems.

Assumption 1: For any subsystem $G_i \in G$ and any condition, c, output by G_i, the following are assumed to hold:

1. c is not an output of any other subsystem.
2. G_i does not output contradictions, and all output conditions of G_i are defined for each marking of G_i.
3. G_i has a finite number of markings.
4. The initial marking of each subsystem is known.

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 many types of systems we consider. However if this assumption is violated, then given initial markings for the subsystems and by item 3 (finite markings) above, we can create one model that represents the composition of the models. Item 2 above states the marking cannot force a condition to be both true and false at the same time.

Currently the observer and taskblock synthesis techniques outlined in this paper require that the subsystem models have a state machine structure, meaning each transition has exactly one input place and exactly one output place. By item 3 above, we know we can convert any of our subsystem models into equivalent condition systems with state machine structures. Thus we can specify a plant model as a generic condition system (i.e. Petri net), and convert it into a state machine structure for synthesis using existing techniques. We have recently implemented this conversion within our Spectool software. We also note that although assumption is logical for subsystem models and for observer models, we do not require it for the taskblock models as synthesized by the methods in [**Error! Reference source not found.**].

The behavior for a plant is described using sequences of condition sets describing the value of input and output conditions of the system. Specifications of desired behavior can similarly be described by such sequences of condition sets, which can include subsets of inputs and outputs, as well as additional conditions. A condition set sequence, called a C-sequence, is a finite length sequence of condition sets. Each condition set sequence is of the form $(C_0C_1...C_k)$. In [**Error! Reference source not found.**] we defined a correspondence between valid marking sequences of our condition systems to C-sequences. Intuitively, a C-sequence corresponds to a marking sequence if the condition set sequence indicates the conditions that are output true by the markings, and if the condition sets indicate the true conditions required for condition enabling of transitions for the marking sequence evolution. We note however that correspondence is not necessarily one-to-one and while our intuitive explanation will suffice for this paper we refer the interested reader to [**Error! Reference source not found.**] for the formal definition.

Define the language $L(\mathcal{G}, m_0)$ to be the set of condition set sequences such that $(C_0C_1C_2...C_n) \in L(\mathcal{G}, m_0)$ if there exists a marking sequence $(m_0...m_k)$ to which it corresponds. The descriptive ordering allows us to describe important characteristics of condition sequences without listing all details of all condition activity within the sequence. Thus, we are allowed to specify high level desired behaviors (the specification) and compare them to sequences that correspond to a marking sequence of the system. Another important aspect of the descriptive ordering is that there can exist an infinite number of equivalent C-sequences which also implies that there exists an infinite number of corresponding C-Sequences given a valid marking sequence. This then implies that the language of the typical condition system contains an infinite number of condition set sequences. We again refer the reader to [**Error! Reference source not found.**] for a formal explanation of the descriptive ordering and for a more detailed discussion of the implications of the descriptive ordering on the language of a condition system.

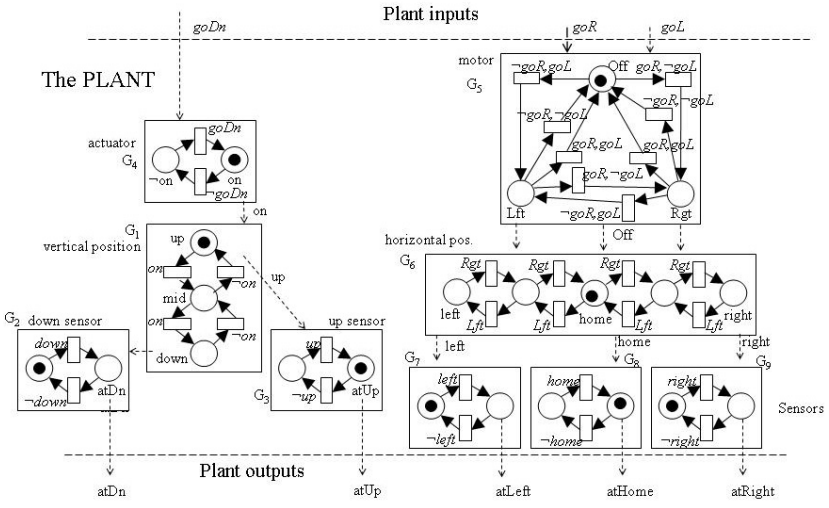

Figure 3: A Condition System Describing the Open-Loop Behavior of a Simple Robotic System

Additionally, not all conditions associated with a condition system are directly observable from the outside of the system. Typically, within an open-loop model of the system only the inputs (actuation signals) and the outputs (sensor readings) are available, and any other condition values must be determined by the observable ones. Obviously, this complicates synthesis procedures and is the motivating issue for the development of observer models.

Example 1. A simple open-loop system is shown in figure 3. The system is composed of 9 subsystem models that communicate via conditions. These models represent the up/down and left/right movement of a simple robotic arm. Places are represented by circles and when marked output an associated condition set. Transitions are represented by bars and input conditions partially determine the model's transition enabling (along with the marking). In this figure, the marking corresponds to an output of *atHome* and *atUp* as *true* and other plant outputs as *false*. The inputs conditions to the plant are *goDn*, *goR* and *goL*. We note, that all other conditions are assumed to be unobservable. Assuming these inputs all had values of false, then we see that the transition with input condition $\neg goDn$ in subsystem G_4 would be state and condition enabled.

We close this section with a brief discussion on the nature of the open-loop models we require for the methods presented in this paper. The models that represent subsystems within a plant might at first appear counter intuitive to researchers within the control engineering community or those familiar with Grafcet. Typically models of this type have output actions associated with places and sensing associated with transitions. However, the plant models that we use have the typical roles of places and transitions reversed since a plant model represents how the system should behave when under control. Thus, when an open-loop system is modeled in such a way then it is possible to synthesize closed-loop controllers for the system that is being modeled. As we shall see the resulting taskblock models have actions associated with places in the usual manner.

3. Synthesis of Observers

An observer is a system that inputs signals from the system and then determines state information of the system. Determining the state information is important because the controllers that we consider depend on knowledge of the state of system subsystems. Just as the control synthesis method focuses on synthesis through local analysis of subsystem models, a state observer for the system can also be synthesized through local analysis of subsystem models. The resulting observers are condition systems also. In [**Error! Reference source not found.**], an observer synthesis method is given for individual subsystems. In [**Error! Reference source not found.**, **Error! Reference source not found.**], it is shown that under certain structural conditions, these individual observers can be interconnected together to generate a state observer for the system, even when some subsystems may not generate any signals that are directly observed.

Generation of a separate observer instead of embedding this information within a controller, as is typically done, offers several potential advantages. First, an observer can be used for things other than control, such as diagnosis and correction, and with observer models this is easily accomplished. It also allows for potential model compactness in that one observer can be used by many taskblocks. And finally it allows for consideration and synthesis of the controller and observer in relative isolation of the other.

In conventional system theory, an *observer* is a system that inputs observations from a plant and outputs an indication of the state of the plant. In this paper, we limit our consideration to observers of subsystem models, but we expand the function of an observer to include estimation of the plant state *and* estimation of unobserved inputs and outputs of the plant. Thus, below we present the problems of condition and state observability defined in [**Error! Reference source not found.**] and [**Error! Reference source not found.**].

First we must introduce a qualitative notion of timing that creates a distinction between *fast* and *slow* transitions and that of a transient state. We again note that the following work presented requires that our subsystem models be represented via a labeled state machine (i.e. - transitions have a single arc in and out, and a single marked place).

Define $T_{\mathcal{G}}^{fast}$ as the set of *fast* transitions for the system $\mathcal{G}$. Transitions in $T_{\mathcal{G}}^{fast}$ will fire without delay. Transitions that are not in $T_{\mathcal{G}}^{fast}$ may (or may not) exhibit delay before firing. Intuitively, transitions in $T_{\mathcal{G}}^{fast}$ would often (but not necessarily) correspond to control decisions or observations (both implemented in software). They can also correspond to transitions in subsystems that react fast with respect to a

physical system (i.e. a sensor or an electronic device). Transitions not in $T_{\mathcal{G}}^{fast}$ (i.e. slow transitions) could correspond to some physical state change (which is assumed to change slower than a state change in software).

Definition 2: Given a time τ and a plant $\mathcal{G}$ with marking m_τ, m_τ is called a *transient marking* at time τ if there exists a fast transition *t* in $T_{\mathcal{G}}^{fast}$ leaving one of the marked places in m_τ that is also condition enabled.

Thus, by this definition a transient marking will change without delay. In our work of observers at this point, we have made assumptions about the nature of the system and observer model. First we assume that observers react faster than the system they control and the taskblocks they output conditions to. The first assumption is natural for observers, and the second can be guaranteed in software. In order for an observer to exist and uniquely identify the state of the plant, the plant must satisfy the properties of state observability and condition observability. First, define C_{obs} as the set of observable conditions for some plant $\mathcal{G}$ (i.e. typically the plant inputs and outputs).

Definition 3: A plant $\mathcal{G}$ is called *state observable* under observed condition set C_{obs} if: Given times τ_0 and τ such that $\tau \geq \tau_0$, if state m_τ is a non-transient state, then m_τ can be determined uniquely from knowledge of initial state m_{τ_0} and the values of conditions in C_{obs} over period τ_0 to τ.

So from the definition, a plant will be state observable if knowledge of an initial state and knowledge of its past observable conditions are sufficient to uniquely determine its current (non-transient) state. The following defines a *condition observable* plant.

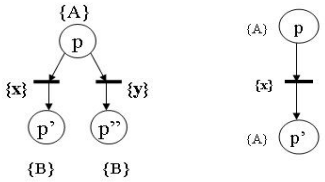

Figure 4: An Example of Direction Confusion and Progress Confusion

Definition 4: A plant $\mathcal{G}$ is *condition observable* for an observed condition set C_{obs} if: Given times τ_0 and τ such that , if marking m_τ is a non-transient state, then the values of all conditions *c* input and output by $\mathcal{G}$ at time τ can be determined uniquely from knowledge of initial state and the values of conditions in C_{obs} over period τ_0 to τ.

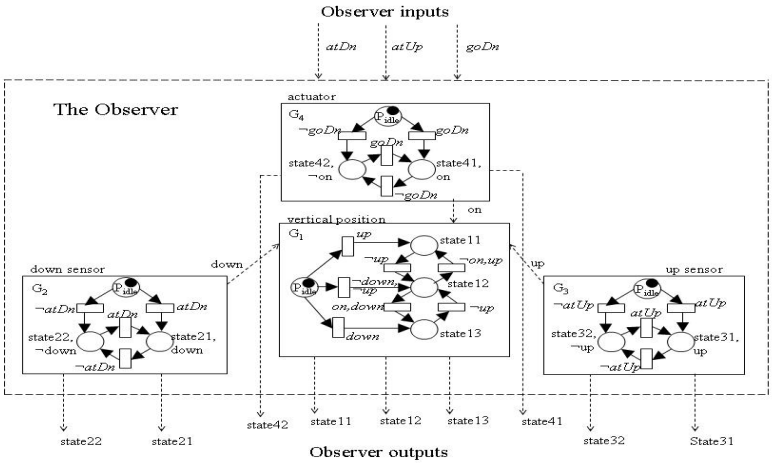

Figure 5: The Observer Model of the Vertical Motion of the Plant

Thus, we know a system is state and condition observable if we can determine the state of the plant given the observation of conditions and knowledge of the initial state. We exempt transient states from this requirement. Two structural properties of condition systems prevent observability though. These are *progress confusion* and *direction confusion*. We refer the reader to figure 4 for examples of each. The precise definitions can be found in [**Error! Reference source not found.**] and [**Error! Reference source not found.**] .

In [**Error! Reference source not found.**] and [**Error! Reference source not found.**], we presented an

algorithm that synthesizes an observer for a single subsystem given lack of progress and direction confusion. We have also shown how to create an observer for an entire system given that each subsystem is condition and state observable.

Example 2: Consider figure 5 which represents the observer models for G_1, G_2, G_3, and G_4, of figure 3. We summarize single layer synthesis with the following. An observer for a subsystem model is created by starting with the original model and replacing outputs with new and unique conditions (one for each place). In figure 5, *State11* is such a condition associated with subsystem G_1. An additional place p_{idle} is added and marked initially. This represents that the state of system is unknown.

The synthesis of an observer is greatly facilitated by places within the subsystem model that are designated as *uniquely marked*. A uniquely marked place has a set of associated output conditions that are only associated with that place. We note that each place in an observer, by synthesis, is uniquely marked. In observer synthesis a transition is added for each uniquely marked place that connects it to the idle state. Another way to determine the state of the system is to identify a sequence of unique outputs and inputs that lead to a marked place that is not necessarily uniquely marked. Although this idea is not covered in this paper, we have considered this in [**Error! Reference source not found.**] and [**Error! Reference source not found.**].

In [**Error! Reference source not found.**], we formally demonstrate classes of system structures which are state observable and condition observable. We also present an observer synthesis method and prove that the synthesized observers correctly estimate the state and unobserved conditions. In [**Error! Reference source not found.**] and [**Error! Reference source not found.**], it was shown that observers for individual subsystems can interact thus allowing for state estimation of subsystems with no directly observed outputs.

In the next section, we present our controller models (taskblocks), discuss the nature of these models, and present a single taskblock synthesized for the example in this paper.

4. Synthesis of Taskblocks

A taskblock is a condition system model that implements control within this framework [**Error! Reference source not found.**][**Error! Reference source not found.**]. A taskblock is synthesized given a specification of desired behavior (a target condition set), and a model representing the observer and subsystem model (the XPlant). In previous work, we have investigated the existence problem of taskblocks given some model of the system (i.e. is a system controllable?). We have also shown how to synthesize taskblocks in a hierarchical and distributed manner given certain restrictions.

Using these methods, we have synthesized compilable C++ code that controlled a modular factory system through a data acquisition board. In that work, we assumed that all states are reachable and the subsystem models were live. As with observer synthesis and its restrictions, we also assume that the subsystem models have a state graph structure. Under these assumptions and given the observability property, we have shown how to synthesize taskblocks that can achieve a target goal independent of the subsystem model's initial state.

Each taskblock has a specific control function. A taskblock becomes *activated* to begin its control function upon its *activation condition*, which uniquely identifies the taskblock. For this paper we denote an activation condition that we will use as *do*. For each element *do* we associate the following:

- *TB*(*do*) is the unique taskblock (condition system model) for which *do* is an input. No other taskblocks or subsystems have *do* as an input.
- *compl*(*do*) is a condition output from the taskblock, indicating task completion associated with a single place within the taskblock.
- *idle*(*do*) is a condition output from the taskblock and indicates that the taskblock is not activated. It is also associated with a single place within the taskblock.
- *sup*(*do*) is a condition input to the taskblock from the supervisory controller. It allows for pre-emption of taskblock execution given the possibility of a forbidden state being achieved.
- $G_{compo}(do)$ is the unique subsystem model associated with the task *do*. The same subsystem model may be associated with many different tasks.

- *goal*(*do*) represents the target condition of the control that is output from the subsystem model. Its value going to "true" signifies task completion.

The interface of the taskblock is shown in figure 6, indicating the signals under discussion.

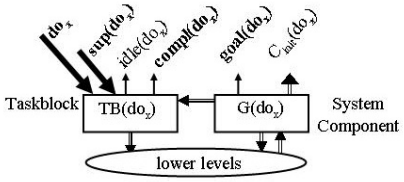

Figure 6: Inputs and Outputs of a Taskblock Associated with Task do_x

In our initial work in the type of supervisory control necessary to determine the truth value of *sup*(*do*), we have added the ability to pre-empt a taskblock only at the beginning of any sequence it may execute [**Error! Reference source not found.**]. Thus we are guaranteed to have a forbidden state avoiding control policy but it is currently non-optimal. We will address this in future research.

The following defines how a taskblock should operate with the rest of the system in order to achieve it's target goal.

Definition 5: For a given activation condition, *do*, and it's associated taskblock, *TB*(*do*), a taskblock is said to be *effective* if it operates as follows:

1. If the taskblock has its activation condition, *do*, become true then the taskblock is said to become *active*. It will remain active as long as the *do* condition remains true.
2. When the taskblock becomes active, then *idle*(*do*) condition signal becomes false.
3. An active taskblock will interact with the subsystem model ($G_{compo}(do)$) (and possibly through other subsystems) in such a manner that it eventually outputs the completion signal *compl*(*do*).
4. When the taskblock outputs the signal *compl*(*do*), then this implies that the associated subsystem $G_{compo}(do)$ is outputting the condition *goal*(*do*) true.

In [**Error! Reference source not found.**], the term *effective* for taskblocks is defined formally using condition languages, and methods were presented to automatically synthesize an effective taskblock *TB*(*do*) by considering only the subsystem $G_{compo}(do)$. The taskblock is activated by a *do* signal associated with some goal condition of the subsystem, and it then outputs activation signals for other taskblocks for lower level subsystems that respond with their own *goal* signals that drive $G_{compo}(do)$.

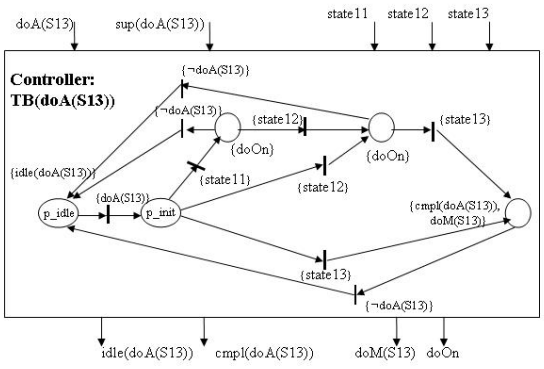

Figure 7: Taskblock Model for Model G_1 Representing a Target of *Down*.

If lower level taskblocks and subsystems are guaranteed to be effective, then (under mild assumptions on the interaction of lower level taskblocks over shared subsystems) the synthesized higher level taskblock can be shown to be effective operating through them.

Because of this result, when synthesizing taskblocks, it is sufficient to abstract away all lower levels into a *Direct Translator*. Simply put, a Direct Translator abstraction represents a set of lower layers of

taskblocks and subsystems as a single system that takes activation signals (such as *do*) for the lower levels and directly returns the corresponding completion signals (such as *compl*(*do*)) to the taskblock and goal signals (such as *goal*(*do*)) to drive the higher layer subsystem. This abstraction of lower layers allows synthesis of any given taskblock to be done efficiently by using information only about single subsystems.

Finally, we have considered taskblock synthesis under partial observation. In this situation, modular observers provide inputs to taskblocks given observations from the plant. An *XPlant* is a synthesized model that represents the simple composition of the subsystem model and the observer. It is used during taskblock synthesis whenever observers are needed, but is not used afterwards. This model is easily synthesized and is used in a straightforward manner for taskblock synthesis. Details can be found in [**Error! Reference source not found.**].

Example 3: The taskblock of figure 7 is synthesized from an Xplant model for subsystem G_1. This taskblock corresponds to a desired behavior of "move down". The observer model G_1 of figure 5 provides the state inputs to this taskblock. The output *goDn* is an input to another taskblock associated with the actuator model G_4 of previous figures.

5. Fault Detection and Diagnosis

In our work, fault diagnosis is the localization of a set of subsystems that could explain a faulty behavior. This is in contrast to the typical diagnosis of traditional DES approaches where diagnosis is the identification of a fault event that would be input to the plant. In this section, we define the detection, diagnosis, and reconfiguration problem within the condition system framework. We will also outline some of the approaches that have been developed, and briefly discuss some areas of current research. Failure diagnosis is an important area of interest within the DES community. It has been investigated within the traditional DES framework in [**Error! Reference source not found.**] and [**Error! Reference source not found.**] among others. In [**Error! Reference source not found.**], the authors consider probability within the context of diagnosis using stochastic Petri net. Timing has been considered in [**Error! Reference source not found.**] [**Error! Reference source not found.**] [**Error! Reference source not found.**], where the authors of [**Error! Reference source not found.**] [**Error! Reference source not found.**] utilize causal networks for diagnosis, and the authors of [**Error! Reference source not found.**] utilize a rules based formalism. One common point in all of these approaches is that information about faulty behaviors is embedded with the plant models.

In our work, we need not explicitly define faults within our open loop models, although we can models faults if so desired. We can accomplish this by comparing what we expect from the system under control (the models of the system) to what we see from the real system (an observed sequence of conditions). A process that performs 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 a fault has been detected. The diagnosis returns a set of potential candidate subsystems that could be the source of the problem. Fault reconfiguration entails reconfiguring the control to work around failed mechanisms within the plant. These have been considered in [**Error! Reference source not found.**][**Error! Reference source not found.**][**Error! Reference source not found.**].

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 $(C_0^R ... C_{k-1}^R C_k^R)$ 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 6: A subsystem is said to have a *fault* if the language of the real subsystem (G_i^R) is not

contained within the language of the corresponding model (G_i^E) of the expected behavior, i.e. $L(G_i^R, m_0^R) \not\subset L(G_i^E, m_0^E)$ is a fault of subsystem G_i^R.

In this paper 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 2: The real and expected systems are made up of subsystem models such that $\mathcal{G}^R = \{G_1^R, ..., G_{n_R}^R\}$ and $\mathcal{G}^E = \{G_1^E ..., G_{n_E}^R\}$ respectively for some positive integers n_R and n_E. For $\mathcal{G}^R$ and $\mathcal{G}^E$ we require:

1. *All subsystems found in the expected model also exist in the physical system with the same outputs:*

2. *The observed outputs of the real system and the model are the same under their initial markings:*

3. *There exists at most a single subsystem fault:* there exists at most one i such that $L(G_i^R, m_0^R) \not\subset L(G_i^E, m_0^E)$.

We specifically note that under assumption 2, 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, $\mathcal{G}^R$. These unmodeled subsystems may influence subsystems with corresponding models of expected behavior through some unmodeled interconnection. Assumption 2 item 3 is the main limitation for this assumption. In practical systems, a failure of one subsystem often leads to other subsystem 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 condition 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.

In [**Error! Reference source not found.**], we showed that the language generated by the system is the intersection of the languages generated by the subsystems. Thus, each subsystem model within a plant imposes a set of constraints on the language of the system. In [**Error! Reference source not found.**] we extended this idea to 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. Intuitively, a system $\mathcal{G}$ that is relaxed with respect to some subsystem G_i can generate a language independent of any constraints imposed by the subsystem. We have shown in [**Error! Reference source not found.**] how to represent a relaxed subsystem model.

Let s_{obs} be an observed C-sequence. A *fault has been detected* if it is determined that $s_{obs} \notin L(\mathcal{G}^E, m_0^E)|_{C_{obs}}$ where $|_{C_{obs}}$ corresponds to the projection onto the observable condition set. Thus a fault is detected if 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 expressed as follows.

Definition 7: Consider an observed C-sequence $s_{obs} = (C_0^R ... C_{k-1}^R C_k^R)$ such that prior to the last condition set observation, the sequence was consistent with the expected behavior, $(C_0^R ... C_{k-1}^R) \in L(\mathcal{G}^E, m_0^E)|_{C_{obs}}$. 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 subsystem 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 by incorporating a fault detector within the taskblock model structure in [**Error! Reference source not found.**] [**Error! Reference source not found.**]. In [**Error! Reference source not found.**] we introduce a method to diagnose faults by approximating $Diagnosis\ (s_{obs})$. In the following example we focus on the direct evaluation of the diagnosis definition.

Example 4: Consider subsystems ($G_5,...,G_9$) from the plant shown in figure 3, and given the following observed sequence (neglecting negated conditions and noting that the motor is off):

s_1 = ({AtLeft}, {AtLeft}{AtLeft, AtHome}).

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 $Diagnosis\ (s_1) = \{G_6, G_8\}$, for the following reasons:

G_5 : $G_5 \notin Diagnosis\ (s_1)$ since under the single fault assumption (Assumption 2 item 3), even if the motor failed, there is no way that the rest of the system could output *AtLeft* and *AtHome* simultaneously.

G_6 : $G_6 \in Diagnosis\ (s_1)$ since if its model behavior were relaxed, it could potentially output both *Left* and *Home*, driving G_7 and G_8 to output *AtLeft* and *AtHome*.

G_7 : $G_7 \notin Diagnosis\ (s_1)$, since even if G_7 was failed, under the single fault assumption, G_6 should still not get to *Home* when the motor is off.

G_8 : $G_8 \in Diagnosis\ (s_1)$ since G_8 could output *AtHome* unexpectedly.

G_9 : $G_9 \notin Diagnosis\ (s_1)$ since it is not outputting a faulty condition valuation.

Next, consider the sequence

s_2 = ({AtLeft,GoR}, {AtLeft,GoR},{AtLeft,GoR, AtHome, AtRight}).

We determine that $Diagnosis(s_2) = \{G_6\}$, since under the single fault assumption, only the fault of G_6 could account for *AtLeft*, *AtHome*, and *AtRight* to be simultaneously true.

We have also considered a best possible diagnosis in [**Error! Reference source not found.**]. In [**Error! Reference source not found.**][**Error! Reference source not found.**], 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 detects and diagnosis of faulty behaviors. Evaluation of definition 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 item 2 determines the set of potentially faulty subsystems. Thus, we are able to directly evaluate these definitions to determine a best possible diagnosis given the constraints of observability.

We have also considered fault reconfiguration in [**Error! Reference source not found.**]. The current method assumes a fault has been detected and diagnosed. Given this information, a subsystem model is modified to capture the faulty behavior (currently transitions are removed which represent a fault that limits functionality). A control is shown to exist in the presence of the fault if there exists a new taskblock (synthesized using existing techniques) given the fault model. Additionally, a faulty subsystem may propagate faulty behaviors to other subsystem thus limiting the functionality of influenced subsystems. We are currently attempting to expand our method to provide a more robust reconfiguration policy and synthesis procedure.

In [**Error! Reference source not found.**], we present an improved method to perform a best possible diagnosis. In that work, we exploit the causal structure to sometimes diagnose a system without the need to directly evaluate Definition 7. We have also recently expanded our fault diagnosis to allow for multiple system failures. This is currently under review. Finally, we are currently investigating the inclusion of fault modeling into this method. Our intention is to give preference to diagnoses that include a-priori fault models. However if an unmodeled fault (or faults) cannot explain the observed behavior then we resort to our original diagnosis procedure.

6. Spectool 2.4 Beta

In this section we will briefly present the software tool, Spectool, used for our research in condition systems. We have recently revised Spectool and the current version, 2.4 Beta, is available for download. The new version of Spectool contains a completely new interface and has been significantly re-coded with the following principal design objectives. Figure 8 is a snapshot that shows how the interface looks.

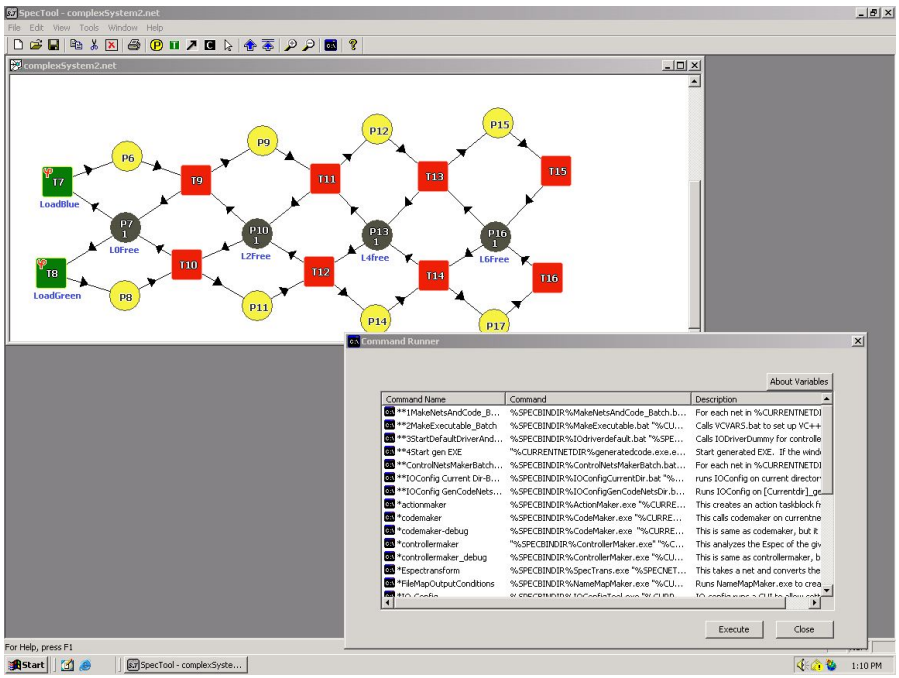

Figure 8: Spectool 2.4 Beta. A Simple Condition System Model Representing a Conveyor System and the Command Runner Interface is Shown.

1. *Utilize the existing NetStructureDLL class library.* NetStructureDLL encapsulates the behavior of a condition system. Therefore all existing control synthesis code (and others) can be directly reused. Microsoft Visual C++ is the development platform.

2. *Improve software robustness.* The goal was to fix memory leaks and improve interface response

especially when viewing and editing large models.

3. *Move towards a standardized file format.* The new Spectool can now read and write nets using the proposed PNML file format standard.[2] This feature is still considered to be beta and will likely evolve as the standard is finalized.

4. *Maintain the separation of graphics and net dynamics.* By maintaining the object-oriented approach to program design and by utilizing the NetStructureDLL class library, a researcher can develop tools without intimate knowledge of programming visual applications. A tool, called Command Runner, makes it easy to develop and implement non-graphical analysis programs into Spectool.

We feel that these objectives have been met. We have maintained backward compatibility and dramatically improved the interface. Large nets can be effectively handled and we have implemented member functions that read and write in the PNML format.

In the NetStructureDLL class library, each condition system is an object and we have created member functions for the creation, analysis, modification of these objects. Within each condition system the places, transitions, arcs, and condition labels are also objects. Constructor functions allow for easy synthesis of new condition system models based on analysis. In the new release, each place and transition object can have their coordinates specified. A system (which may be composed of plant models, control models, among other things) is represented as a collection of these objects.

For those familiar with object-oriented programming using C++, it is relatively easy to write analysis code using the NetStructureDLL class library. Spectool can also be used for analysis and synthesis for sub-classes of condition systems including automata, state machines with guards, a class of Petri Nets, and a class of interpreted Petri Nets.

Spectool can be downloaded at *http://www.engr.uky.edu/~holloway/spectool/*. Example Microsoft Visual C++ workspaces using the condition system library are included in the distribution. We have included a conversion program in the spirit of the algorithm presented in [**Error! Reference source not found.**] that translates a condition system (as a labeled Petri Net) into a coverability tree and a finite state machine (if one exists).

7. Discussion

In this paper, we have presented an overview of our research into automated control and fault diagnosis using models of interacting discrete event subsystems. In particular, we have summarized earlier work in control synthesis, observer synthesis, and diagnosis methods.

The research described in this paper is intended to develop a collection of synergistic methods that use reusable, interacting component-level models. While we have considered many problems within this framework, there exist many opportunities for future research. First, timing information can be valuable both in synthesis and in simulation, and the methods presented should be extended to use timing information. Second, in some systems, the state space of individual subsystems could be very large, and some early work in representing such large state spaces through colored Petri net techniques has been done in [**Error! Reference source not found.**]. Further work in colored models could be used for improved control and diagnosis methods that exploit the structure of such models.

Another area of important research that we have not considered is simulation and verification methods to ensure the validity of subsystem models used, as well as techniques for improving modeling using the condition systems framework. Conversion of alternative models into the condition system framework, and vice-versa, would be useful for bringing analysis methods and tools from other frameworks to our synthesis toolbox. The relation of condition languages with CTL models has been initially considered in [**Error! Reference source not found.**].

2 Please see the following URL for details on the PNML standard: *http://www2.informatik.hu-berlin.de/top/pnml/about.html*.

Finally, we should emphasize that our investigations have been motivated by a number of applications and scenarios in factory automation and embedded systems control. Letting applications drive the research directions will continue to uncover new frontiers for extending this research.

REFERENCES

1. 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.

2. ASHLEY, J. and L. E. HOLLOWAY, **A fault detection scheme using taskblocks**. In Proceedings of CDC-ECC'05: 44th Int. Conf. on Decision and Control and European Control Conference, Seville, Spain, December 2005.

3. ASHLEY, J. and L. E. HOLLOWAY, **Exploiting causal structure in the refined diagnosis of condition systems.** In Proceedings of IEEE-ETFA'06: 11th IEEE International Conf. on Emerging Technologies and Factory Automation., Praha, CR, September 2006.

4. ASHLEY, J., L. E. HOLLOWAY, and N. DANGOUMAU, **Fault recovering taskblocks and control synthesis for a class of condition systems**. In 16th IFAC World Congress., Praha, July 2005.

5. ASHLEY, J. and L.E. HOLLOWAY, **An Equivalent CTL Formulation for Condition Sequences**. Discrete Event Dynamic Systems, 15(4):333–348, 2005.

6. ASHLEY, JEFFREY, **Diagnosis of Condition Systems**. PhD thesis, University of Kentucky, 2004.

7. ASHLEY, JEFFREY and LAWRENCE E. HOLLOWAY, **Qualitative diagnosis of condition systems.** Discrete Event Dynamic Systems: Theory and Applications, 14(4):395–412, 2004.

8. CASSANDRAS, C. and S. LAFORTUNE, **Introduction to discrete event systems**. Kluwer Academic Publishers, 2001.

9. YU GONG. **Observer synthesis for control of a class of condition systems**. PhD thesis, University of Kentucky, 2003.

10. YU GONG and L. E. HOLLOWAY, **State observer synthesis for a class of condition systems**. In IEE International Workshop on Discrete Event Systems (WODES2000), Ghent, August 2000.

11. YU GONG and L. E. HOLLOWAY, **Multi-layer state observers for condition systems**. In 8th IEEE International Conference on Emerging Technologies and Factory Automation(ETFA 2001), Antibes, France, October 2001.

12. 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.

13. XIAOYI GUAN, **Distributed Supervisory Control of Forbidden Conditions, and Automated Synthesis and Composition of Task Controllers**. PhD thesis, University of Kentucky, October 2000.

14. HANISCH, H.-M., A. LUDER, and M. RAUSCH,**Controller synthesis for net condition/event systems with a solution for incomplete state observation**. European Journal of Control, 3(4):280–291, 1997.

15. HOLLOWAY, L. 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.

16. HOLLOWAY, L. E. and J. ASHLEY, **Elaborative orderings of condition languages**. In Proceedings of 1998 IEEE Conference on Decision and Control, Tampa, December 1998.

17. HOLLOWAY, L. E., ANDY CALLAHAN, JOHN O-REAR, and XIAOYI GUAN, **Spectool: Automated synthesis of control code for discrete event controllers.** In Discrete Event Systems: Analysis and Control. Kluwer Academic Publishers, 2000.

18. HOLLOWAY, L. E., YU GONG, and JEFF ASHLEY, **State observability and condition observability for a class of interacting discrete event systems**. Mathematics and Computers in Simulation, 70(5):275–286, 2006.

19. .HOLLOWAY, L. E., X. GUAN, R. SUNDARAVADIVELU, and J. ASHLEY, **Automated synthesis and composition of taskblocks for control of manufacturing systems.** IEEE Transactions on Systems, Man and Cybernetics: Part B, 30(5):696–712, October 2000.

20. HOLLOWAY, L. E., XIAOYI GUAN, RANGANATHAN SUNDARAVADIVELU, and JEFF ASHLEY, **Automated synthesis and composition of taskblocks for control of manufacturing systems.** IEEE Trans. On Systems, Man, and Cybernetics Part B., 30(5), October 2000.

21. HUANG, Z., 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.

22. KOWALEWSKI, S., 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. IEEE.

23. PRAVEEN MANDAVILLI**, Colored condition systems**. Master's thesis, University of Kentucky, 2004.

24. PROVAN, GREGORY 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.

25. PROVAN, GREGORY 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.

26. RAMADGE, P. J. and W. M. WONHAM, **Supervisory control of a class of discrete-event processes**. SIAM J. on Control and Optimization, 25(1):206–230, January 1987.

27. SAMPATH, M., R. SENGUPTA, S. LAFORTUNE, K. SINNAMOHIDEEN, and D. TENEKETZIS, **Diagnosability of discrete event systems**. IEEE Transactions on Automatic Control, 40(9):1555–1575, September 1995.

28. SAMPATH, MEERA, 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.

29. SREENIVAS, R.S. and B.H. KROGH, **On condition/event systems with discrete state realizations**. Discrete Event Dynamic Systems: Theory and Applications, 1(2):209–236, May 1991.

30. WONHAM, W. M. and P. J. RAMADGE, **On the supremal controllable sublanguage of a given language**. SIAM J. on Control and Optimization, 25(3):637–659, May 1987.