Proceedings of the
44th IEEE Conference on Decision and Control, and
the European Control Conference 2005
Seville, Spain, December 12-15, 2005

A fault detection scheme using condition systems

Jeff Ashley and Lawrence Holloway

Abstract— **In this paper, fault detection is examined using condition systems. The method presented relies on existing results for controller synthesis using these models. First we present a generalized discussion of the controller module called a taskblock. Then a method to incorporate on-line fault detection into the taskblock framework is presented. We show that the resulting modified controller module is effective for control and detection.**

I. INTRODUCTION

In this paper, we consider fault detection using condition system models introduced by [1]. This modified form of a Petri net allows for distributed and hierarchical modeling where separate models communicate via *condition signals*. In our work, we represent elements or subsystems of an open loop system (called *components*) using this framework. We also use condition systems to implement our component controllers called taskblocks.

In [2] we presented a method to synthesize taskblocks to control some component. These taskblocks can then be used to generate control code. We also showed under what situations these taskblocks could be combined to drive an entire system to some target state. In this paper, we introduce a method to add fault detection capability to taskblocks. This work also builds upon that presented in [3] and [4].

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

Fault detection is a sub-problem of the classical diagnosis problem in discrete event systems research as investigated by [5], [6] and many others. Whereas diagnosis is the identification of the faulty component(s) given observability and other constraints, detection is the acknowledgment that a fault has occurred, and does not include identifying the faulty component. Diagnosis typically includes, by definition, fault detection.

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

J. Ashley is a Research Scientist at University of Kentucky's Center for Manufacturing. ashley@engr.uky.edu

L. Holloway is a Professor of Electrical Engineering and Director of the Center for Manufacturing, University of Kentucky, Lexington, Kentucky 40506, USA holloway@engr.uky.edu

One objective of this paper is to generalize the discussion about taskblock synthesis in an effort to simplify the discussion and to refine how a taskblock would behave in the presence of potentially faulty behaviors. The nature of taskblocks as presented in [2] has been modified in the process. The main 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 paper is organized as follows. First we will review condition systems and taskblocks. Next we present a generalized description of taskblock synthesis and the limitations we assume for this paper. We then present a fault detection scheme, and show that the desired behavior of the taskblock is preserved and that we do in fact detect faulty behaviors.

II. CONDITION SYSTEMS

In this paper, 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 (here called *components*) as well as the interaction of a system with a controller [2].

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 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 paper, 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 $m \in M_{\mathcal{G}}$, $g_{\mathcal{G}}(m) = \{c | \exists p \text{ s.t. } c \in \Phi_{\mathcal{G}}(p) \text{ and } m(p) \geq 1\}$, where $g_{\mathcal{G}}(m)$ is the set of output conditions forced "true" by marking m.

0-7803-9568-9/05/$20.00 ©2005 IEEE

3) *Next-state dynamics depend on state enabling and condition enabling:* for any $m \in M_{\mathcal{G}}$ and any $C \subseteq$ *AllC* of conditions with value "true", $m' \in f_{\mathcal{G}}(m, C)$ if and only if there exists some transition set T such that
   a) T is *state-enabled*, meaning $(\forall p \in \mathcal{P}_{\mathcal{G}})\ m(p) \geq |\{t \in T | p$ is input to $t\}|$
   b) T is *condition-enabled*, meaning $(\forall t \in T)\ \Phi_{\mathcal{G}}(t) \subseteq C$
   c) the next marking m' satisfies $\forall p \in \mathcal{P}_{\mathcal{G}}$, $m'(p) = m(p) - |\{t \in T |\ p$ is input to $t\}| + |\{t \in T |\ p$ is output of $t\}|$
4) $M_{\mathcal{G}}$ *is closed under* $f_{\mathcal{G}}(\cdot)$*:* if $m \in M_{\mathcal{G}}$ and $m' \in f_{\mathcal{G}}(m, C)$ for some $C \subseteq$ *AllC*, then $m' \in M_{\mathcal{G}}$.

We define the output condition set for a system $\mathcal{G}$ as $C_{out}(\mathcal{G}) = \{c \in \Phi_{\mathcal{G}}(p) | p \in \mathcal{P}_G\}$. Similarly, define $C_{in}(\mathcal{G}) = \{c \in \Phi_{\mathcal{G}}(t) | t \in \mathcal{T}_{\mathcal{G}}\}$. Note that a condition system can be subdivided into components, where each component is a condition system over a set of connected places and transitions which are disconnected from all other places and transitions. For the remainder of this paper we will use the notation $\mathcal{G}$ to indicate the complete system, and the notation $\{G_1, \ldots G_n\}$ to indicate the set of components in $\mathcal{G}$. Given an initial marking m_0 of $\mathcal{G}$, we let $m_{0,i}$ denote the marking over the places in $G_i \in \mathcal{G}$.

We also make the following assumption on the structure of our components.

Assumption 1: For any component $G_i \in \mathcal{G}$ and any condition $c \in C_{out}(G_i)$, the following are assumed to hold:

1) c *is not an output of any other component:* $\forall j \neq i$, $c \notin C_{out}(G_j)$.
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 component, G_i. 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.

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 $(C_0 C_1 \ldots C_k)$ for some integer k and sets $C_i \subseteq$ *AllC* for all $0 \leq i \leq k$. A set of C-sequences is called a language, and the language consisting of all C-sequences is denoted $\mathcal{L}$. A C-sequence is also the mechanism used for specifying the desired behavior of some system used for controller synthesis. Details can be found in [2].

The empty condition set, $\emptyset$, is important in specifying desired behavior for controller generation. It represents a "don't care" condition in a C-sequence meaning we don't care what output (via conditions) a system takes in completion of some task under direction. We will also assume that some of 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, $C_i \neq C_{i+1}\ for\ 1 \leq i \leq k-1$. We note that we can easily make any such C-sequence trim by repetitively removing repetitive condition sets.

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 "expected" system is denoted by $(C_0^E \cdots C_{k-1}^E C_k^E)$ and the initial marking by m_0^E.

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. $L(G_i^R, m_{0,i}^R) \not\subseteq L(G_i^E, m_{0,i}^E)$. A fault is detected if the observed condition sequence from the real system is not contained within the expected language of the system, i.e. $s_{obs} \notin L(\mathcal{G}^E, m_0^E)$ [3].

III. A BLOCK DIAGRAM PERSPECTIVE OF TASKBLOCKS.

In [2] we were interested in using an open-loop system composed of component models to develop controllers that would drive a system to a targeted state. In that work, these controllers are generated by a set of connected taskblocks that are generated by analysis of the component models. We did not consider faulty behavior and so the basic assumption was that the real system would behave exactly as expected (i.e. $L(G_i^E, m_0^E) = L(G_i^R, m_0^R)$).

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 of real system components be denoted by $\mathcal{G}_{compo}^R$.

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 $\mathcal{G}^R \subseteq \mathcal{G}_{compo}^R \cup \mathcal{G}_{tasks}$.

Each taskblock has a specific control function. Let x denote the intended control function. In [2], such a control function may represent either driving the system to a given condition or controlling the system to maintain a condition as $true$. A taskblock becomes *activated* to begin its control function upon its *activation condition*, which uniquely identifies the taskblock. Let $C_{do} \subset$ *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.
- $goal(do_x) \in C_{out}(G_{compo}(do_x))$ 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 $G_{compo}(do_x)$.

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).

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 2: 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 $s \in L(\mathcal{G}^E, m^E_0)$, if $(\emptyset\{\neg do_x\}) \leq s$, for any set $C_{ext} \subseteq$ *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 $ss' \in L(\mathcal{G}^E, m^E_0)$, $(C_{ext}) \leq s'$, 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 1: Figure 1 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 $G_{compo}(do^A_{mid})$. (The superscript "A" in the condition do^A_{mid} will be explained in section IV). The component model represents a robot position as either arm_up, mid, or arm_down. To move the arm requires conditions m_up or m_down, indicating motor on up and motor on down. We have $goal(do^A_{mid}) = mid$.

The taskblock has an initial state with p_{idle} marked and condition output $idle(do^A_{mid})$. Upon receipt of the activation

signal do^A_{mid}, the marking changes and is no longer idle. Since the component has place p_3 marked and outputs condition arm_down, then the taskblock marking moves from p_{idle} to p_{init} to p_8. From that state, the taskblock outputs condition $do^A_{m\_up}$. If that condition is effective for G_{sys}, then by definition 4, G_{sys} eventually outputs condition $goal(do^A_{m\_up}) = m\_up$. 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 $compl(do^A_{mid})$, indicating completion.

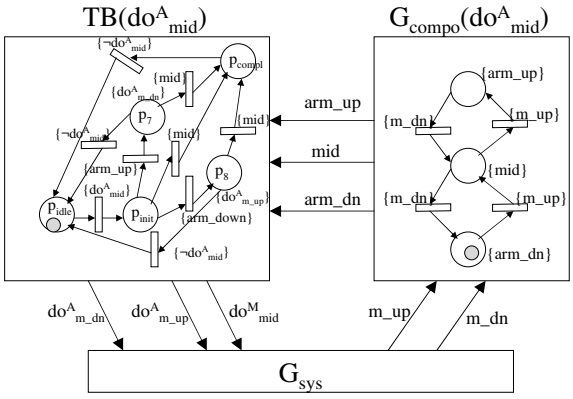

Fig. 1. An example of effective do_x.

In our analysis of taskblocks, it is important for us to assume that the controller reacts faster than the component that it controls. In [2] we defined a *control wait state* as a marking within a component model such that the marking cannot be changed without some condition input change. This leads to the *Prompt Controller Assumption* which guarantees that a controller can react fast enough to direct a component in a desired manner.

Finally, we 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 violated under the presence of a faulty behavior as we shall see.

Definition 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'.

IV. A GENERALIZED DISCUSSION OF TASKBLOCK SYNTHESIS

In this section, we consider the modeling details of taskblocks. In [2] 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^A_x is the activation condition for the action-type taskblock $TB(do^A_x)$ with $goal(do^A_x) = x$, and do^M_x is the activation condition for the maintain-type taskblock $TB(do^M_x)$ with $goal(do^M_x) = x$. The following assumptions will define the nature of taskblocks for the remainder of this paper and hold for both action-type and maintain-type taskblocks. As a result, we will not use the superscript notation in the following discussion.

First we to define the following. Define, the full condition mapping for some condition system as $\mathcal{C}^\star_G := \bigotimes_{c \in C_{out}(G)} \{c, \neg c\}$ where $\bigotimes$ is defined as the cross product. Thus, for example, if non-negated outputs of G are $\{c_1, c_2, c_3\}$ then $\mathcal{C}^\star_G = \{c_1, \neg c_1\} \times \{c_2, \neg c_2\} \times \{c_3, \neg c_3\}$.

In [2] 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.

Assumption 2: Taskblock Structure Assumption (TSA): Given an activation condition, do_x, its 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 $C \in \mathcal{C}^\star_{G_{compo}(do_x)}$.
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 behavior from 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.

V. 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 (i.e. a fault).

For fault detection, we will assume that under execution of some task do_x will be output continually. Thus fault detection occurs under an activated taskblock. 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, except under behavior that is not "expected".

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. First, define $fault(do_x)$ as a fault condition that is associated with a single new place within a taskblock.

Definition 4: Given a system $\mathcal{G}^E \subseteq \mathcal{G}_{tasks} \cup \mathcal{G}^E_{compo}$ with initial state m_0^E and a condition $do_x \in C_{in}(\mathcal{G}^E) \cap C_{do_x}$ such that $idle(do_x) \in g(m_0^E)$, we define that do_x is *effective for detection* for $\mathcal{G}^E$ under m_0^E if :

Unexpected component behavior while taskblock activated implies eventual fault: Given any $(C_1 \cdots C_k C_{k+1}) \in L(\mathcal{G}^R, m_0^R)$ such that:

1) Observations are in the expected language up to the k'th observation from the component, or $(C_1 \cdots C_k) \in L(\mathcal{G}^E, m_0^E)$,
2) At the $k+1$ observation of the component, an unexpected behavior is encountered, or $(C_1 \cdots C_k C_{k+1}) \notin L(\mathcal{G}^E, m_0^E)$, and
3) for any C_{k+2} consistent with the taskblocks, i.e. $(C_1 \cdots C_k C_{k+1} C_{k+2}) \in L(\mathcal{G}_{tasks}, m_0^E)$, with $do_x \in C_{k+2}$, then $fault(do_x) \in C_{k+2}$, so the taskblock indicates a fault state.

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 and effective for detection 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 1.

Definition 5: Given a taskblock $TB(do_x)$ not effective for detection define for all $p \in \mathcal{P}_{TB(do_x)} - \{p_{idle}, p_{compl}\}$, the set $CSet_{fault}(p)$ as:

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

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 paper that we do in fact detect deviations between expected and real behavior, and that the input/output behavior of the original taskblock is preserved. Let $TBF(do_x)$ denote a transformed taskblock.

Algorithm 1: An algorithm to create $TBF(do_x)$ from $TB(do_x)$.

Input: do_x, $TB(do_x)$ that is not effective for detection, $G_{compo}(do_x)$
Output: $TBF(do_x)$
Copy $TB(do_x)$ as new taskblock $TBF(do_x)$.
Create a place, p_{fault}, Assign output condition $fault(do_x)$ to this place.
$\forall p \in \mathcal{P}_{TB(do_x)} - \{p_{idle}, p_{compl}, p_{fault}\}$
{
  $\forall C \in CSet_{fault}(p)$
  {
    Add new transition, t', with C as the enabling condition for this transition, assign input place to t' place p, and the ouput place is p_{fault}.
  }
}

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). Another efficient solution might be found by simply checking a changed condition set input to a taskblock against all of the expected responses and if it is not one of these then it must be a fault. This is a subject of future research.

The following result shows that our new taskblock that detects fault behaviors is effective for control and effective for detection.

Theorem 1: Given do_x, a taskblock $TB(do_x)$ satisfying the taskblock stucture assumption(TSA) but that is not effective for detection, the component model $G_{compo}(do_x)$, and a $TBF(do_x)$ constructed via algorithm 1, do_x is *effective for control* and *effective for detection*.

Proof: By assumption 2 (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.* $\diamond\diamond\diamond$

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 is if one of the unexpected responses leads the system to the fault state.

Example 2: Consider p_{init} and p_8 from figure 1 as shown in figure 2. This figure shows the expansion of these places that occur to construct $TBF(do_{mid})$ from $TB(do_{mid})$. To make the figure easier to read, we represent the input conditions arm_up as up, and arm_dn as dn.

We would also add additional outgoing transitions to the fault state for p_7 in a manner similar to p_8. We also note that the transitions associated with the deactivation condition for the taskblock (i.e. $\neg do^A_{mid}$) would remain unchanged in the $TBF(do_{mid})$. We have omitted these in the figure, and p_{idle} for clarity.

VI. Discussion

In this initial paper 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.

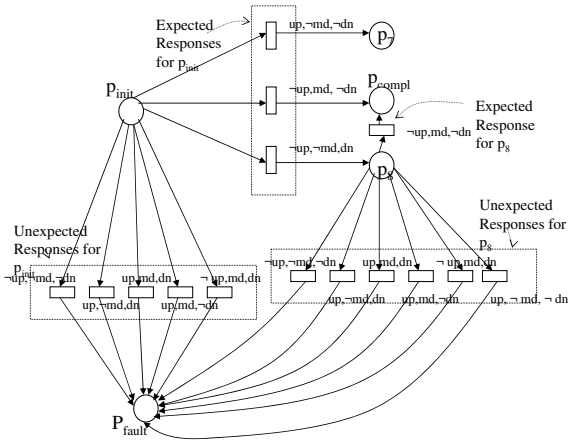

Fig. 2. The transformation of p_{init} and p_8 from figure 1 to the new places in $TBF(do_x)$.

We have also included a generalized discussion of taskblock structures and tied this work into the ideas of the "real" and "expected" systems. We also note that we do not require that faulty behaviors be encoded within the component model, instead we automatically determine these by the $TBF(do_x)$ transformation. Although, we can model faults if desired.

We envision that a method such as this could be used in conjunction with a diagnosis method such as the one presented in [3] 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 $TBF(do_x)$; finding what types of component models lend themselves to taskblock synthesis; and introducing timing into this framework. We are also working on implementing these ideas on fault detection in the Spectool software tool detailed in [2].

References

[1] R. Sreenivas and B. Krogh, "On condition/event systems with discrete state realizations," *Discrete Event Dynamic Systems: Theory and Applications*, vol. 1, no. 2, pp. 209–236, May 1991.

[2] L. E. Holloway, 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*, vol. 30, no. 5, pp. 696–712, October 2000.

[3] J. Ashley and L. E. Holloway, "Qualitative diagnosis of condition systems," *Discrete Event Dynamic Systems: Theory and Applications*, vol. 14, no. 4, pp. 395–412, 2004.

[4] J. Ashley, "Diagnosis of condition systems," Ph.D. dissertation, University of Kentucky, 2004.

[5] M. Sampath, R. Sengupta, S. Lafortune, K. Sinnamohideen, and D. C. Teneketzis, "Diagnosability of discrete event systems," *IEEE Transactions on Automatic Control*, vol. 40, no. 9, pp. 1555–1575, 1995.

[6] A. Aghasaryan, E. Fabre, A. Benveniste, R. Boubour, and C. 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*, vol. 8, no. 1, pp. 203–231, 1998.