FAULT RECOVERING TASKBLOCKS AND CONTROL SYNTHESIS FOR A CLASS OF CONDITION SYSTEMS.

Jeff Ashley[1], Lawrence E. Holloway[1], Nathalie Dangoumau[2]

[1] *Dept. of Electrical Engineering and Center for Manufacturing,*
University of Kentucky, Lexington, Kentucky 40506, USA
email: holloway, ashley@engr.uky.edu
and
[2] *LAGIS - CNRS - UMR 8146*
Ecole Centrale de Lille
BP 48, F 59651 Villeneuve d'Ascq Cedex, France
email: Nathalie.Dangoumau@ec-lille.fr

Abstract: In this paper, we define fault recovery in terms of condition system languages, and show how to modify a component model to represent a fault that limits the functionality of a component. We also show under what conditions a control can be synthesized to work around such a fault. Finally, we consider the propagation of faulty behavior throughout the system and present an algorithm to evaluate whether some target specification is achievable by utilizing existing control synthesis techniques. *Copyright ©2005 IFAC*

Keywords: Discrete-event systems, Petri-nets, Fault-tolerant systems, Fault diagnosis, Control system synthesis, Intelligent manufacturing systems

1. INTRODUCTION

In this paper, we consider fault recovery using condition system models introduced by (Sreenivas and Krogh, 1991). This modified form of a Petri net allows for distributed and hierarchical modeling where separate models communicate via *condition signals*. Among other things, we have studied condition systems in the problems of control synthesis (Holloway *et al.*, 2000) and fault diagnosis (Ashley and Holloway, 2004) for classes of condition systems. Here, we consider a method to recover from such faults. Our initial approach uses an off-line technique to generate a control module (called a *taskblock*) given some faulty behavior. First, we define fault recovery in terms of condition system languages, and show how to model faults that limit the functionality of a component. We then show that if a taskblock exists, we can achieve a target condition set even in the presence of a fault. Finally we consider the propagation of faulty behavior throughout the system and present an algorithm to evaluate whether some target specification is achievable. A discrete event systems approach to fault recovery has also been investigated in (Moosaei and Zad, 2004).

Of course in using an off-line method, it is very helpful to know the fault a-priori and this presents a problem in that it either: requires exact knowledge of failure modes; or, leads to a potentially huge set of control modules needed to account for every possible fault. This may be alleviated to some extent by consideration of techniques found in (Dangoumau and Craye, 2003).

2. 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 (Holloway *et al.*, 2000).

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 $\mathcal{C}$ be the universe of all conditions, such that for each condition c in $\mathcal{C}$, 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 $f_\mathcal{G} : M_\mathcal{G} \times 2^\mathcal{C} \longrightarrow 2^{M_\mathcal{G}}$, and a condition output mapping $g_\mathcal{G} : M_\mathcal{G} \longrightarrow 2^\mathcal{C}$. 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 $(\forall p)\Phi_\mathcal{G}(p) \subseteq \mathcal{C}$ maps output conditions to each place, and $(\forall t)\Phi_\mathcal{G}(t) \subseteq \mathcal{C}$ maps *enabling conditions* to each transition. The net is related to $M_\mathcal{G}$, $f_\mathcal{G}$, and $g_\mathcal{G}$ in the following manner:

(1) *The states are the markings of the Petri net:* each state $m \in M_\mathcal{G}$ is a function over $\mathcal{P}_\mathcal{G}$ that represents a mapping of non-negative integers to places.
(2) *The output conditions have their truth value established by marked places:* for any $m \in M_\mathcal{G}$,
$g_\mathcal{G}(m) = \{c \mid \exists p \text{ s.t. } c \in \Phi_\mathcal{G}(p) \text{ and } m(p) \geq 1\}$, where $g_\mathcal{G}(m)$ is the set of output conditions forced "true" by marking m.
(3) *Next-state dynamics depend on state enabling and condition enabling:* for any $m \in M_\mathcal{G}$ and any $C \subseteq \mathcal{C}$ 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 $(\forall p \in \mathcal{P}_\mathcal{G})$ $m(p) \geq |\{t \in T \mid p$ 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 $\forall p \in \mathcal{P}_\mathcal{G}$, $m'(p) = m(p) - |\{t \in T \mid p \text{ is input to } t\}| + |\{t \in T \mid p \text{ is output of } t\}|$
(4) *$M_\mathcal{G}$ is closed under $f_\mathcal{G}(\cdot)$:* if $m \in M_\mathcal{G}$ and $m' \in f_\mathcal{G}(m, C)$ for some $C \subseteq \mathcal{C}$, then $m' \in M_\mathcal{G}$.

We define the output condition set for a system $\mathcal{G}$ as $C_{out}(\mathcal{G}) = \{c \in \Phi_\mathcal{G}(p) \mid p \in \mathcal{P}_G\}$. Similarly, define $C_{in}(\mathcal{G}) = \{c \in \Phi_\mathcal{G}(t) \mid t \in \mathcal{T}_\mathcal{G}\}$. 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 $\mathcal{G}$. 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.
(3) G_i is a state graph with one token.

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 $C_i \subseteq \mathcal{C}$ 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 $\mathcal{L}$. 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.

3. TASKBLOCKS

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 $\mathcal{G}_{compo}$. 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.

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. Let $C_{do} \subset \mathcal{C}$ be the set of *activation conditions* associated with taskblocks. For each element $do \in C_{do}$ we associate the following:

- $TB(do) \in \mathcal{G}_{tasks}$ is the unique taskblock (condition system model) for which $do \in C_{in}(TB(do))$. No other taskblocks or components have *do* as an input.
- $compl(do) \in C_{out}(TB(do))$ is a condition output from the taskblock, indicating task completion.
- $idle(do) \in C_{out}(TB(do))$ is a condition output from the taskblock and indicates that the taskblock is not activated.
- $G_{compo}(do) \in \mathcal{G}_{compo}$ is a component model associated with the task *do*. The same component model may be associated with many different tasks.
- $goal(do) \in C_{out}(G_{compo}(do))$ is a condition output from the component model.
- $C_{init}(do) \subseteq C_{in}(TB(do)) \cap C_{out}(G_{compo}(do))$ is a set of *initiation conditions* for the taskblock that are output from the component $G_{compo}(do)$.

For a given activation condition *do* and its 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 while its initial conditions are true (all $c \in C_{init}(do)$ are 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 its $idle(do)$ condition signal becomes false.
(3) An active taskblock will interact with the component model ($G_{compo}(do)$) (and possibly through other components) 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 component $G_{compo}(do)$ is outputting the condition $goal(do)$ true.

In (Holloway *et al.*, 2000), 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 component $G_{compo}(do)$. The taskblock is activated by a *do* signal associated with some goal condition of the component, and it then outputs activation signals for other taskblocks for lower level components that respond with their own *goal* signals that drive $G_{compo}(do)$. If lower level taskblocks and components are guaranteed to be effective, then (under mild assumptions on the interaction of lower level taskblocks over shared components) 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 components as a single system that takes activation signals (such as do_x) for the lower levels and directly returns the corresponding completion signals (such as $compl(do_x)$) to the taskblock and goal signals (such as $goal(do_x)$) to drive the higher layer component. This abstraction of lower layers allows synthesis of any given taskblock to be done efficiently by using information only about single components.

In this paper, we are not concerned with the details of a taskblock synthesis method, so we consider a function to represent such a method. In the definition of such a function below, we consider that the specification for the closed-loop behavior of the taskblock and its component is given as a c-sequence $s = (C_{init}, \emptyset, C_{goal})$. Figure 1 shows how taskblocks and components interact. Note that in the figure, DT_i for $i = 1, 2, 3, 4$ represents the three Direct Translators required for this system. We note that for this paper, the *specification net* represents the top-level taskblock that has a target condition set equal to the singleton set $\{c_{doSpec}\}$.

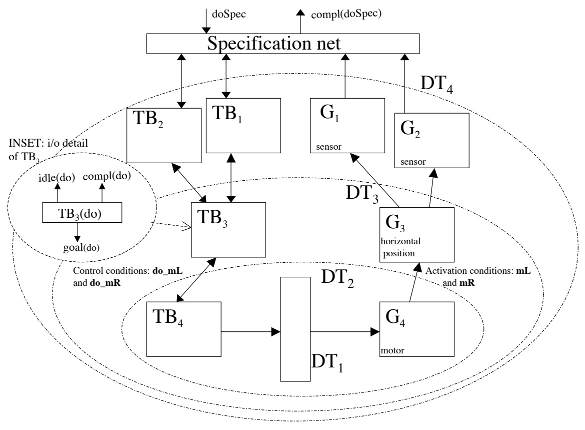

Fig. 1. The condition system and controller.

Definition 2. Given some component model G, an assumed initial condition set C_{init} (possibly empty, indicating no requirement), and a goal condition set $C_{goal} \subseteq C_{out}(G)$, define $MakeControl(G, C_{init}, C_{goal})$ as the set of pairs $(TB, C_{TB,init})$ where TB is a taskblock effective for G for goal conditions C_{goal} under true initial conditions $C_{TB,init}$, where $C_{init} \subseteq C_{TB,init} \subseteq C_{out}(G)$.

The statement $C_{init} \subseteq C_{TB,init}$ means that a taskblock returned by the function can expect more stringent initial condition requirements than the specification required. Note that $MakeControl(G, C_{init}, C_{goal})$ can return an empty set, meaning that no effective control exists. In (Holloway *et al.*, 2000), the procedure presented generated pairs in $MakeControl$ of the form $(TB, \emptyset)$ which implies that the taskblock will work from any initial state for the component. Here, we also assume that a taskblock must also work from any initial state.

Finally, we define a *steerable path* within a component G_i implies the ability of some controller (captured by some taskblock and some direct translator) to enable required transitions and to prevent other transitions from firing in order to eventually reach C_{goal}.

4. COMPONENT LEVEL FAULT RECOVERY

In this paper, we assume there exists a special set of conditions called *assumables* (Ashley, 2004). We refer to this set of assumables as C_{asmbl}, where the set C_{asmbl} must be contradiction free. Assumables are not outputs of any component. Rather, they are inputs which are not directly observed. They are used to represent assumptions of correct operation of the system. Under normal operation, each $c \in C_{asmbl}$ has value true. We model a fault as an assumable condition that has become false.

When an assumable becomes false, thus representing a fault, then any transitions that rely on it for enabling cannot become enabled, thus changing the dynamics of the component which has the assumable as an input. Some transitions may also have the negation of the assumable as a condition input. Such transitions are normally disabled, but could become enabled if the assumable becomes false. In either case, any resulting change in the reachability of the component will change the possible combinations of condition values that can be output by the component. This then can prevent the enabling of transitions in other components, preventing other conditions from becoming true, and propagating the fault through other components.

In this section, we consider that we are given a set of condition *sets* that are forced false. This could include a singleton set with an assumable, or it can include sets of conditions that are outputs of other subsystems.

Definition 3. Define a *minimal set of condition sets failed false*, $C_{fail,F} \subset 2^{\mathcal{C}}$, such that:

(1) For any $C \in C_{fail,F}$, the conditions in C cannot be simultaneously true under any reachable marking, given c_{asmbl} is false;
(2) For any $C \in C_{fail,F}$, there does not exist a strict subset $C' \subset C$ that is also in $C_{fail,F}$.

For example, for a system with conditions a and b, the occurrence of a fault may result in the set $\{a, \neg b\}$ being in the failed false set. Thus, the system could output a and b both true, both false, or a false and b true. However, a true with b false could not occur, and so any transition with condition input including $\{a, \neg b\}$ cannot be enabled to fire. Finally, we note that any condition c that will be forced true from the fault can be represented by having $\{\neg c\} \in C_{fail,F}$.

The following defines a transform of a component model so it captures the behavior of some given fault.

Definition 4. Given a component G_i , the sets of conditions $\mathcal{C}_{fail,F}$ failed false then transform the component as follows. Define this new component as $G(G_i, \mathcal{C}_{fail,F})$.

(1) For each transition $t \in T_i$ such that $C \subseteq \Phi(t)$ for some $C \in \mathcal{C}_{fail,F}$, then remove t.
(2) For each transition $t \in T_i$ such that $\Phi(t) = \bigcup \neg c$ for some distinct singleton sets $\{c\}$ in $\mathcal{C}_{fail,F}$, then for each $t' \in p^{(t)}$ for $p \in {}^{(p)}t$ where $t' \neq t$ and $\Phi(t') \neq \bigcup \neg c$ for some distinct singleton sets $\{c\}$ in $\mathcal{C}_{fail,F}$ then remove t'.

Item 1 represents removing transitions that cannot be enabled due to condition sets that are failed false. For item 2, under a fault some transitions may become enabled (and cannot be disabled). Since we cannot prevent these t from firing then we don't want to explore paths through t', since we cannot be guaranteed to steer the system through these transitions.

Next we show that a component model operating under some failed false input conditions, will behave the same as the transformed component model resulting from applying definition 4.

Lemma 1. Given a component G_i, a marking m, a target set C_{goal}, and a set of conditions $\mathcal{C}_{fail,F}$, there exists a steerable path to achieve C_{goal} from m in $G(G_i, \mathcal{C}_{fail,F})$ if and only if there exists a steerable path to achieve C_{goal} from m in G_i under $\mathcal{C}_{fail,F}$.

Proof:

Only if : Any steerable path in $G(G_i, \mathcal{C}_{fail,F})$ from m that achieves C_{goal} must exist in G_i since application of definition 4 to create $G(G_i, \mathcal{C}_{fail,F})$ only removes transitions from G_i. Since the path exists in $G(G_i, \mathcal{C}_{fail,F})$ it must also exist in G_i.

If: First note that all transitions t removed from G_i in creation of $G(G_i, \mathcal{C}_{fail,F})$ are either: *a*) disabled in G_i since they have an condition enabling $\Phi(t)$ that is a superset of some C in $\mathcal{C}_{fail,F}$; or *b*) cannot be forced to fire in G_i since there exists a transition t' leading from its input place such that t' is always condition enabled. In case a, these transitions cannot be enabled and so there cannot exist a steerable path to C_{goal} from m through any of these transitions in G_i under $\mathcal{C}_{fail,F}$. In case b, since t' cannot be disabled we cannot be guaranteed to fire t, and hence we cannot have a steerable path through t. So if there exists some steerable path to achieve C_{goal} in G_i from m under $C_{fail,F}$ then it does not include any transitions disabled by items a) and b) above. Since creation of $G(G_i, C_{fail})$ only removes these transitions then it follows that the steerable path must also exist in $G(G_i, \mathcal{C}_{fail,F})$. $\diamond\diamond\diamond$

Thus any C_{goal} that cannot be reached would be failed false. The following immediately follows from the previous lemma and definition 3.

Corollary 1. Given a component G_i, a marking m, and a set of conditions $\mathcal{C}_{fail,F}$, then the set of all C_{goal} sets non-reachable from marking m in $G(G_i, \mathcal{C}_{fail,F})$ have a corresponding minimal set of condition sets failed false.

This then characterizes how faulty behaviors may propagate throughout a system. Thus a faulty component that is unable to output certain target condition values may propagate faulty behaviors to other components (which may not have failed, but nevertheless act faulty). The next lemma shows extends these ideas to the entire system within a language framework. Let $\mathcal{G}_{compFlt}$ be equal to the new system created by replacing G_i with $G(G_i, C_{fail,F})$ in $\mathcal{G}_{compo}$. Define $L_{control}(\mathcal{G}, s, M_0)$ as the set of all C-sequences of $\mathcal{G}$ that satisfied control specification $s = (C_{init}, \emptyset, C_{goal})$ given a set of markings M_0.

Lemma 2. Given some system $\mathcal{G}_{compo}$, a transformed system $\mathcal{G}_{compFlt}$, a set of initial markings M_0, and a control specification $s = (C_{init}, \emptyset, C_{goal})$, then the following statements are true.

(1) $L_{control}(\mathcal{G}_{compFlt}, s, M_0) \subseteq L_{control}(\mathcal{G}_{compo}, s, M_0)$.
(2) If $L_{control}(\mathcal{G}_{compFlt}, s, M_0) = \emptyset$ then there does not exist an effective control policy that will achieve the desired C-sequence s.
(3) If $L_{control}(\mathcal{G}_{compFlt}, s, M_0) \neq \emptyset$ then there exists an effective control policy that will achieve the desired specification s.

Proof: To prove item 1, consider the language generated by G_i and $G(G_i, C_{fail,F})$. It is obvious that the language of $L(G_i, m)$ for all $m \in M_0$ contains more or at least all of the sequences in $L(G(G_i, C_{fail,F}), m)$ for all $m \in M_0$. Or, $L(G(G_i, C_{fail,F}), m) \subseteq L(G_i, m)$. By Theorem 1 from (Ashley and Holloway, 2004), then it follows that $L(\mathcal{G}_{compFlt}, m) \subseteq L(\mathcal{G}_{compo}, m)$ for all $m \in M_0$. Since a control C-sequence s restricts the behavior of $\mathcal{G}_{compo}$ to a set of possible outputs consistent with its target, and that it does so consistently across $\mathcal{G}_{compFlt}$ and $\mathcal{G}_{compo}$, we see that $L_{control}(\mathcal{G}_{compFlt}, s, M_0) \subseteq L_{control}(\mathcal{G}_{compo}, s, M_0)$. Items 2 and 3 follow directly from item 1. $\diamond\diamond\diamond$

The next theorem ties Lemma 1 to the notion of an effective taskblock that can steer a faulty component around disabled transitions. In particular, it shows if there exists a set of initial conditions and a taskblock in $MakeControl(G(G_i, C_{fail,F}), C_{init}, C_{goal})$ then they also exist in the set of taskblocks for the original component G_i under the fault.

Theorem 1. Given some component G_i, a set of conditions $C_{fail,F}$ failed false, the transformed component $G(G_i, C_{fail,F})$, a set of initial conditions C_{init}, a target set of conditions C_{goal}, if there exists some pair $(TB, C_{TB,init}) \in MakeControl(G(G_i, C_{fail,F}), C_{init}, C_{goal})$ then $(TB, C_{TB,init}) \in MakeControl(G_i, C_{init}, C_{goal})$.

Proof: If $(TB, C_{TB,init})$ exists in $MakeControl(G(G_i, C_{fail,F}), C_{init}, C_{goal})$ then that implies that there exists some steerable path in $G(G_i, C_{fail,F})$ that achieves C_{goal}. It immediately follows from lemma 1, that the same steerable path exists in G_i and so $(TB, C_{TB,init}) \in MakeControl(G_i, C_{init}, C_{goal})$. $\diamond\diamond\diamond$

5. PROPAGATION OF FAULTS IN THE CONDITION SYSTEM AND FAULT RECOVERY.

As previously noted, a faulty component may affect other components thereby making them behave in a faulty manner. This propagation of faults then affects multiple components in the system and may make a top-level control objective c_{doSpec} unachievable. However, if there exists redundancy or means within a component to work around a fault, then will exist a sequence of control inputs that can still achieve c_{doSpec} even in the presence of a fault. In this section, we present a method to propagate faults through a system and show when such a fault prevents c_{doSpec} from being realized by the system.

The following algorithm propagates the influence of faulty condition sets throughout the system. First, note C_{fail}^{ALL} contains all elements of C_{fail}^{j} for all j. C_{fail}^{j} is the set of all failed false conditions up to iteration j in the algorithm. C_{Δ}^{j} is the set of all failed conditions added at iteration j in the algorithm.

Algorithm 1. Given system $\mathcal{G} = \{G_1, G_2, \ldots, G_n\}$, a set of assumable conditions $C_{fail,F}$ failed false associated with a single G_i.

```
(1)  $j \Leftarrow 0$
(2)  $C^j_{fail} \Leftarrow C_{fail,F}$ and $C^j_\Delta \Leftarrow C_{fail,F}$
(3)  While $C^j_\Delta \neq \emptyset$ :
(4)    $C^{j+1}_{fail} \Leftarrow C^j_{fail}$
(5)    $C^{j+1}_\Delta \Leftarrow \emptyset$
(6)    For each $G_k$ s.t. $C \cap C_{in}(G_k) \neq \emptyset$ for some $C \in C^j_\Delta$:
(7)      For each $C' \in 2^{|C_{out}(G_k)|}$ in set of possible target condition sets output by $G_k$ :
(8)        If $\nexists$ a $(TB, \emptyset) \in MakeControl(G(G_k, C^j_{fail}), \emptyset, C')$ then :
(9)          Add $C'$ to $C^{j+1}_{fail}$.
(10)         Add $C'$ to $C^{j+1}_\Delta$.
(11)         Minimize $C^{j+1}_{fail}$, $C^{j+1}_\Delta$ using definition 3.
(12)   $j \Leftarrow j + 1$
(13) $C^{ALL}_{fail} \Leftarrow C^j_{fail}$
```

We are ready to present the final result of this paper. The following proof states that if our top level specification is in C^{ALL}_{fail} then it is not achievable under the conditions presented.

Theorem 2. Given some top-level control specification $\{C_{init}, \emptyset, \{c_{doSpec}\}\}$, a condition set C^{ALL}_{fail} resulting from applying Algorithm 1 for some $\mathcal{G}$ and $C_{fail,F}$, if c_{doSpec} is in C^{ALL}_{fail} then c_{doSpec} is not achievable given any arbitrary marking $m \in \mathcal{M}$.

Proof: We prove by induction on the index j. For $j = 0$: $C^0_{fail} = C_{fail,F}$ and any $C \in C_{fail,F}$, by definition, is unachievable.

For $j > 0$: For some $j \geq 0$, suppose that all $C \in C^j_{fail}$ implies C is not achievable. Then consider some $C' \notin C^j_{fail}$ but $C' \in C^{j+1}_{fail}$. By the algorithm lines 8 and 9, C' is only added to C^{j+1}_{fail} if no taskblock exists with $(TB, \emptyset) \in MakeControl(G(G_k, C^j_{fail}), \emptyset, C')$ for the G_k that outputs the condition set C'. Thus, $C' \in C^{j+1}_{fail}$ is not achievable. By the induction, if $\{c_{doSpec}\}$ is in C^{ALL}_{fail}, then c_{doSpec} is not achievable. $\diamond\diamond\diamond$

Our current taskblock synthesis method for the class of systems we consider are guaranteed to be able to complete a task given any initial condition set for the associated component (Holloway *et al.*, 2000). In as much, our current taskblock synthesis method is consistent with this proposed algorithm. However, this statement may obviously be violated under a fault, and so generating a controller under the restriction of a required set of initial conditions is an important subject of future research.

6. DISCUSSION

We have presented and defined fault recovery for the condition system/taskblock framework based on the languages of these systems. For a class of faults that limit functionality, we show how to modify a component model to account for this behavior. If a new taskblock does not exist for this modified component then there in no way to achieve the target for the component. This in turn can propagate faulty behavior to other components. If this propagation continues to the top-level specification net, then we show that the top-level target in unachievable given that the system can start from any state.

7. ACKNOWLEDGMENTS

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.

REFERENCES

Ashley, Jeffrey (2004). Diagnosis of Condition Systems. PhD thesis. University of Kentucky.

Ashley, Jeffrey and Lawrence E. Holloway (2004). Qualitative diagnosis of condition systems. *Discrete Event Dynamic Systems: Theory and Applications* **14**(4), 395–412.

Dangoumau, N. and E. Craye (2003). Modeling for reconfiguration of production systems: Multipoint of views. In: *31st International Conference on Computers and Industrial Engineering).* San Francisco, CA.

Holloway, L. E., X. Guan, R. Sundaravadivelu and J. Ashley (2000). Automated synthesis and composition of taskblocks for control of manufacturing systems. *IEEE Transactions on Systems, Man and Cybernetics: Part B* **30**(5), 696–712.

Moosaei, M. and S. Hashtrudi Zad (2004). Fault recovery in control systems: A modular discrete-event approach. In: *Proc. 2004 International Conference on Electrical and Electronics Engineering (CINVESTAV / IEEE).* Acapulco, Mexico. pp. 445–450.

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