MedCrave
Step into the World of Research

International Robotics & Automation Journal

Review Article

Open Access

CrossMark

Timed fault-tolerant supervisory control

Abstract

In Mulahuwaish,[1–4] we investigated the problem of fault tolerance in the framework of untimed discrete-event systems (DES). This approach is different from the typical fault-tolerant methodology as the approach does not rely on detecting faults and switching to a new supervisor; it requires a supervisor to work correctly under normal and fault conditions. This is a passive approach that relies upon inherent redundancy in the system being controlled. In this paper we extend the work of Mulahuwaish[1–4] to the timed DES (TDES) setting. We introduce our setting, and then provide a set of timed fault tolerant definitions designed to capture different types of fault scenarios and to ensure that our system remains controllable in each scenario. As the nonblocking property is the same for timed and untimed DES, the untimed fault-tolerant nonblocking properties and algorithms from Mulahuwaish[1–4] can also be used in the timed setting without any changes. We then present algorithms to verify these properties followed by complexity analyses and correctness proofs of the algorithms. An example is then provided to illustrate our approach.

Keywords: discrete-event systems, supervisory control, fault-tolerant

Volume 10 Issue 2 - 2024

Aos Mulahuwaish,[1] Amal Alsuwaidan,[2] Ryan J Leduc[3]

[1]Department of Computer Science and Information Systems, Saginaw Valley State University, USA
[2]King Abdulaziz City for Science and Technology, King Abdullah Rd, Saudi Arabia
[3]Department of Computing and Software, McMaster University, Canada

Correspondence: Ryan J Leduc, Department of Computing and Software, McMaster University, 1280 Main St. West, Hamilton, ON, Canada, L8S 4K1, Email leduc@mcmaster.ca

Received: May 25, 2024 | **Published:** June 13, 2024

Introduction

Supervisory control theory, introduced by Ramadge and Wonham,[5–7] provides a formal framework for analysing discrete-event systems (DES). In this theory, automata are used to model the system to be controlled and the specification for the desired system behaviour. The theory provides methods and algorithms to obtain a supervisor that ensures the system will produce the desired behaviour.

However, the base theory typically assumes that the system behaviour does not contain faults that would cause the actual system to deviate from the theoretical model. An example is a sensor that detects the presence of an approaching train. If the supervisor relies on this sensor to determine when the train should be stopped in order to prevent a collision, it could fail to enforce its control law if the sensor failed.

In Mulahuwaish[1,3,4] we introduced a discrete-event system-based fault tolerance approach that was designed to handle intermittent faults. An intermittent fault is a malfunction of a device or system that occurs at intervals, usually irregular, in a device or system that functions normally at other times. A loose connection is an example of this kind of fault.

In the above approach, we introduced uncontrollable fault events to the system's plant model and then categorized some common fault scenarios. By scenarios, we refer to several common fault situations that we would want our supervisors to be able to handle. The scenarios range from simple situations that are easy to verify (for example, at most one faults are allowed to occur), to ones that are more flexible in the occurrence of faults, but more expensive to verify. We then developed some properties that allowed us to determine if a supervisor will still be controllable and nonblocking in these scenarios. We note that this is a passive approach that relies upon inherent redundancy in the system being controlled.

In this paper, we will extend the work of Mulahuwaish[1–4] to the timed DES (TDES) setting.[8–10] Timed DES extends untimed DES theory by adding a new *tick* (τ) event, corresponding to the tick of a global clock. The event set of a TDES contains the *tick* event as well as other non-tick events called activity events (Σ_{act}). This is a powerful extension as TDES adds to untimed DES the ability to express when an event is possible, when it must occur by (possibly infinite upper bound), and the ability to force certain events (called forcible events) to occur in a specified time frame (before the next clock tick). As TDES is more expressive, both in modelling and enforcement, extending fault-tolerant supervisors to the TDES setting clearly will be useful.

The primary difference between our timed and untimed fault-tolerant results is that the tick event must not be a fault event, and that the controllability condition for TDES differs from the untimed setting. We thus have to adapt the fault-tolerant definitions and algorithms to use the timed controllabilty definition (which also ensures forcing of events is done properly). Fortunately, verifying nonblocking (a weak check to make sure the system does not deadlock or livelock) is the same for both timed and untimed DES so we don't have to develop timed nonblocking fault-tolerant properties; we can simply re-use the fault-tolerant nonblocking properties and algorithms developed in Mulahuwaish.[1–4]

Illustrative example

We now introduce an example to illustrate our method. We will briefly introduce the example here, and then use it to explain the various aspects of our approach as we introduce them. After we have fully introduced our method, we will provide the remaining portions of the example in Section 7, and then discuss the results of applying our approach to the example.

Example setting

Our example is based on the manufacturing testbed from Leduc.[11] The testbed was designed to simulate a manufacturing workcell using model train equipment, in particular problems of routing and collision. Figure 1 shows conceptually the structure of the full testbed and sensors.

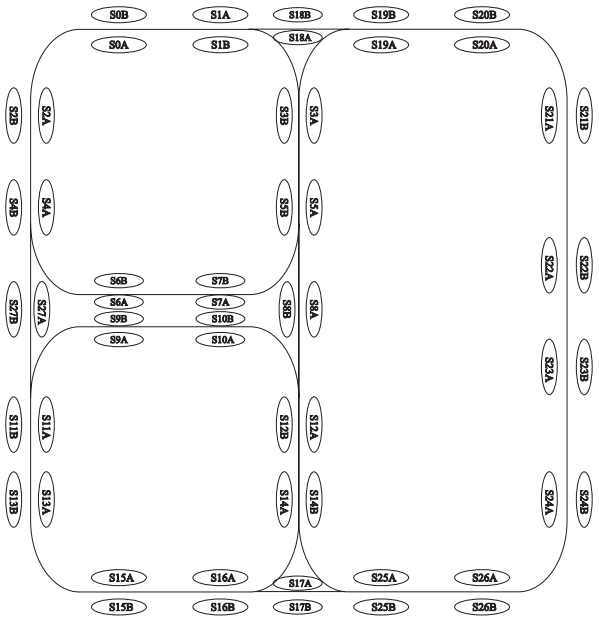

Figure 1 Sensors in the testbed.

We will initially focus on only a single track loop, shown in Figure 2. The loop contains eight sensors and two trains (*train 1, train 2*). Train 1 starts between sensors 9 and 10, while train 2 starts between sensors 15 and 16. Both trains can only traverse the tracks in a counter clockwise direction.

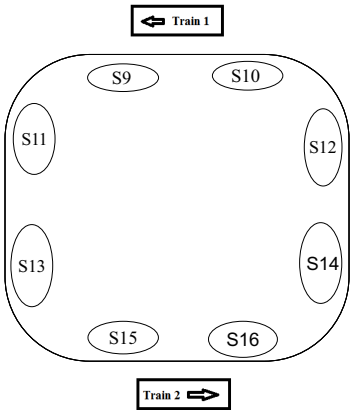

Figure 2 Single train loop.

The sensor models, shown in Figure 3, indicate when a given train is present, and when no trains are present. Also, they state that only one train can activate a given sensor at a time. The figure shows the original sensor model, one for each sensor $J \in \{9, \ldots, 16\}$, before fault events were added to the plant model.

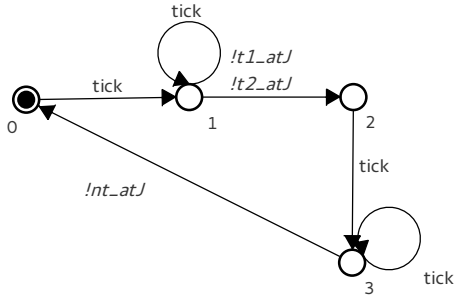

Figure 3 Sensor $J = 11,\ldots,15$.

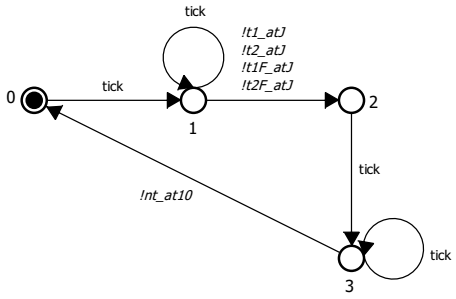

Figure 4 Sensors $J = 9,10,16$ with fault events.

Figures 5 and 6 show the sensor's interdependencies with respect to a given train. With respect to the starting position of a particular train (represented by the initial state), sensors can only be reached in a particular order, dictated by their physical location on the track. Both DES already show the added fault events.

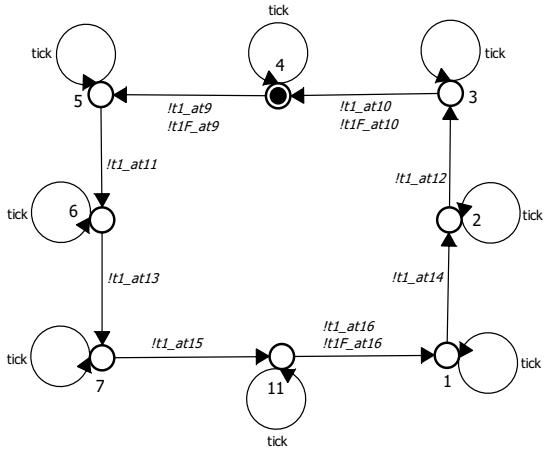

Figure 5 Sensor interdependencies for train 1.

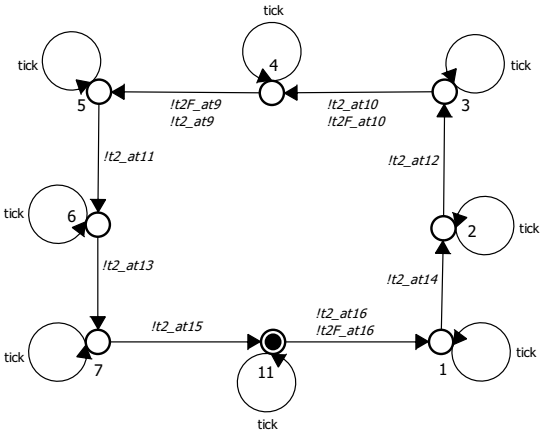

Figure 6 Sensor interdependencies for train 2.

We note that in the DES diagrams, circles represent unmarked states, while filled circles represent marked states. Two concentric, unfilled circles represent the initial state. If the initial state is also marked, the inner circle is filled. Uncontrollable events are indicated by an "!" preceding the event's name, such as "*!t1_atJ*".

Adding intermittent faults

To add faults to the model, we assumed that sensors 9, 10, and 16 could have an intermittent fault; sometimes the sensor would detect the presence of a train, sometimes it would fail to do so.

Citation: Mulahuwaish A, Alsuwaidan A, Leduc RJ. Timed fault-tolerant supervisory control. *Int Rob Auto J.* 2024;10(2):48–61.
DOI: 10.15406/iratj.2024.10.00283

We modelled this by adding to all the plant models a new event $t1f\_atJ$, $J \in \{9, \ldots, 16\}$, for each $t1\_atJ$ event. For each $t1\_atJ$ transition in a plant model, we added an identical $t1f\_atJ$ transition. The idea is we can now get the original detection event or the new fault one instead. We made similar changes for train 2. Figure 4 shows the new sensor models with the added fault events. We note that the fault events must be uncontrollable events as it would be unrealistic if supervisor could simply disable a fault event and prevent the fault from occurring.

Now consider the problem of preventing a second train from entering the track segment bounded by sensors 11 and 13, when this section is already occupied by the first train. Ideally, we would monitor sensor 10 for the arrival of the second train, and halt that train until the first train has left the protected track segment. However, if sensor 10 faulted, the train would not stop and we would have a collision. We could make our controller more redundant by monitoring both sensors 9 and 10, and we could then safely stop the train as long as both sensors did not fail. In such a situation, we could tolerate a single fault, but not two in a row.

We further note that we cannot allow our supervisor to make decisions based on the occurrence of the sensor fault events as we cannot realistically expect such faults to be observable. The supervisor must only change its control actions based on observing non fault events.

Literature review

Currently in the DES literature, the most common approach when a fault is detected is to switch to a new supervisor to handle the system in its degraded mode. Such an approach focuses on fault recovery as opposed to fault tolerance. This requires the construction of a second supervisor, and requires that there be a means to detect the occurrence of the fault in order to initiate the switch. In our approach, we use a single supervisor that will behave correctly for the original system without faults, and for the system with added fault events that are restricted to the fault scenarios that we are addressing. This is a passive approach that relies on the inherent redundancy in the system being controlled. Our method has the advantage that we only need to design a single supervisor for our system, and that we do not need to detect that a fault has occurred for our approach to work. We will now discuss some relevant, related work.

Two closely related topics to fault-tolerance and fault recovery are robust and adaptive supervisory control as discussed by.[12–14] In both approaches, the system G of interest is not specified exactly, but either belongs to a set of possible plants, or we are given a set of "lower" and "upper" bounds. For robust control, the goal is to construct a supervisor that will achieve a desired behavior for all of the possible plants. This is 6 analogous to our passive approach to fault-tolerance.

Adaptive control, on the other hand, monitors system behavior and uses the information to resolve or reduce the uncertainty in the system's behavior in order to improve the performance of the system. This is analogous to active fault recovery methods. It is worth noting that both methods involve synthesis, where our approach is based on user designed supervisors and verification. As synthesis algorithms have higher complexity than verification algorithms,[15] our approach should be applicable to larger systems. Also, modular supervisors are typically easier to understand and implement than the results of synthesis.

An additional drawback with active fault recovery methods is that they require that a fault be detected, and possibly identified if there are multiple faults, before the fault recovery response can be applied. Constructing a fault diagnoser can be expensive,[16] and has the additional concern that it may not detect the fault in time to respond appropriately. As our approach is passive and can handle the original and faulted system, response time is not a concern for us. However, the tradeoff is that our approach may result in an overly cautious supervisor.

While adaptive and robust control are related, neither has a concept of fault events and thus cannot be used directly for fault-tolerance or recovery as their supervisors could be designed to take action on the occurrence of a fault event which should be unobservable to supervisors. However, methods such as Saboori et al.,[14] which make use of partial observations, could perhaps be adapted by setting fault events to be unobservable, and using a model without faults, and a post-fault model.

This of course raises the question of how the post-fault model would be obtained. Simply adding fault events to an existing model often results in a system with strings that contain so many faults in them that no controllable and nonblocking supervisor would exist. Where it is true they could make use of the models generated by our approach, but then robust/adaptive control would be unnecessary as synthesis could just be done directly on the resulting model as there would be no uncertainty left.

Finally, it might be possible to use robust/adaptive control on the original plant model without fault events, and new post-fault models without fault events. However if the system contains multiple faults, generating separate models for each possible post fault system (i.e. system behavior after a specific sequence of faults have occurred) could be tedious, error prone, and time consuming. Our approach on the other hand, uses a single system model with all faults already added. We provide a simple approach and methodology for adding faults to an existing system model that could be easily automated

Qin Wen et al.,[17] introduces a framework for fault-tolerant supervisory control of discrete-event systems. In this framework, plants contain both normal behavior and behavior with faults, as well as a submodel that contains only the normal behavior. The goal of fault-tolerant supervisory control is to enforce a specification for the normal behavior of the plant and to enforce another specification for the overall plant behavior. This includes ensuring that the plant recovers from any fault within a bounded delay so that after the recovery, the system state is equivalent to a state in the normal plant behavior. They formulate this notion of fault-tolerant supervisory control and provide a necessary and sufficient condition for the existence of such a supervisor. The condition involves notions of controllability, observability and relative-closure together with the notion of stability.

In Paoli et al.,[18] they propose to detect faults and switch to a different supervisor before the nominal system behaviour is violated. The controller is updated based on the information provided by online diagnostics. The supervisor needs to detect the malfunctioning component in the system in order to achieve the desired specification. The authors propose the idea of safe diagnosability as a step to achieve fault-tolerant control.

In Park et al.,[19] they present necessary and sufficient conditions for fault-tolerant robust supervisory control of discrete-event systems that belong to a set of models. When these conditions are satisfied, fault-tolerance can be achieved based on the identification of tolerable fault sequences. In the paper, the results were applied to the design, modelling, and control of a workcell consisting of arc welding (GMAW) robots, a sensor, and a conveyor.

Citation: Mulahuwaish A, Alsuwaidan A, Leduc RJ. Timed fault-tolerant supervisory control. *Int Rob Auto J.* 2024;10(2):48–61.
DOI: 10.15406/iratj.2024.10.00283

Brandin et al.,[8–10] added a new dimension to the basic DES theory by introducing timed discrete-event systems (TDES). They introduced the concept of a global clock and tick event. Also, they introduced the ability to specify when certain events must occur.

Research has been conducted to discuss faults in the TDES setting. However, this research focused on fault recovery and fault detection, as opposed to fault tolerance.

In,[20] the main goal of Allahham et al.,[20] was to detect system faults as early as possible. Their proposed idea was to construct a TDES with two clocks: one clock would reflect the task state and and the other clock would measure the elapsed time since the task had been started. They assumed that each task had normal behavior with no faults, and acceptable behavior with intermittent faults within a bounded delay. Their approach was to give each task a time interval. Then, they would check if the task had finished in the defined time interval or before it, which means the system had no faults or it had intermittent faults that the system can tolerate. They monitored the TDES with stopwatch automaton that modeled the acceptable behavior for a specific task. The stopwatch had three locations: initial, normal execution, and interruption, to specify the task status.

In, Moosaei et al.,[21] introduced fault recovery to TDES. Their system consists of the plant and a diagnosis system, both modeled using activity transition graphs (ATG). The plant model describes its behavior in both normal and faulty conditions. The diagnosis system was assumed to be available to detect and isolate faults whenever they occurred. They have introduced three modes for their system: normal when no faults occur, transient when a fault occurs, and recovery when the fault was detected and isolated. Their design consists of a normal-transient supervisor, and multiple recovery supervisors for each failure mode.

As we will see in the following section, our approach is quite different to the preceding methods. Rather than focus on synthesis approaches, ours is based on verification. We assume that the designer has used their understanding of the given system and its possible faults to attempt to design a supervisor that is controllable and nonblocking for the system both without faults, and when faults occur according to our specified scenarios. Our goal is to provide a method to verify if they have achieved this.

Overview

This paper is organized as follows. Section 1 provides an introduction to our topic. Section 2 discusses DES preliminaries. Section 3 introduces fault events and the fault scenarios to which they apply. Section 4 presents our timed fault-tolerant controllability definitions. Section 5 presents algorithms to verify the timed fault-tolerant controllability properties and provides a complexity analysis. Section 6 presents algorithm correctness proofs and Section 7 provides a small manufacturing example to illustrate our approach. Finally, Section 8 provides conclusions and future work.

Preliminaries

We now present a summary of the DES terminology that we use in this paper. For more details, please refer to.[22,34]

Strings and languages

Let Σ be a finite set of distinct symbols (events). Let Σ^+ denote the set of all finite, non-empty sequences of events, and Σ^* be the set of all finite sequences of events including $\in$, the empty string. We can then define $\Sigma^* : \Sigma^+ \cup \{\in\}$. For $s \in \Sigma^*, |s|$ equals the length (number of events) of the string.

Let $L \subseteq \Sigma^*$ be a language over Σ. A string $t \in \Sigma^*$ is a prefix of s $s \in \Sigma^*$ (written $t \le s$) if $s = tu$, for some $u \in \Sigma^*$. The prefix closure of language L (denoted $\overline{L}$) is defined as $\overline{L} := \{t \in \Sigma^* | t \le s \text{ for some } s \in L\}$. Let $\text{Pwr}(\Sigma)$ denote the set of all possible subsets of Σ. For language L, the eligibility operator, $\text{Elig}_L : \Sigma^* \to \text{Pwr}(\Sigma)$, is given by $\text{Elig}_L(s) := \{\sigma \in \Sigma | s\sigma \in L\}$ for $s \in \Sigma^*$.

Timed DES

Timed DES (TDES)[8–10] extends untimed DES theory by adding a new tick (τ) event, corresponding to the tick of a global clock. The event set of a TDES contains the tick event as well as other non-tick events called activity events (Σ_{act}).

A TDES automaton is represented as a 5-tuple $\mathbf{G} = (Q, \Sigma, \delta, q_o, Q_m)$ where Q is the state set, $\Sigma = \Sigma_{act} \dot{\cup} \{\tau\}$ is the event set, the partial function $\delta : Q \times \Sigma \to Q$ is the transition function, q_o is the initial state, and Q_m is the set of marker states. We extend δ *to* $\delta : Q \times \Sigma * \to Q$ in the natural way. The notation $\delta(q,s)!$ means the transition is defined. The closed behavior of G is defined to be $L(\mathbf{G}) := \{s \in \Sigma^* | \delta(q_o, s)!\}$. The marked behavior is defined as $L_m(\mathbf{G}) := \{s \in L(G) | \delta(q_o, s) \in Q_m\}$.

The reachable state subset of DES $\mathbf{G}$, denoted Q_r, is: $Q_r := \{q \in Q | \exists s \in \Sigma^*) \delta(q_o, s) = q\}$. A DES G is reachable if $Q_r = Q$. We will always assume that a DES is reachable, has a finite state and event set, and is deterministic (single initial state and at most a single transition leaving a given state for a given event).

TDES contain forcible (Σ_{for}), and prohibitable events (Σ_{hib}) Forcible events are non-tick events which can be relied upon to preempt tick, when needed. The method used by a TDES supervisor to indicate that an event $\sigma \in \Sigma_{for}$ or should be forced (made to occur before the next tick) at a given state, is to "disable" tick at this state. This has the effect of removing the now impossible behavior that tick could occur before σ. Prohibitable events are non-tick events that can be disabled. The set of controllable events are $\Sigma_c = \Sigma_{hib} \dot{\cup} \{\tau\}$, and the uncontrollable events are $\Sigma_u = \Sigma - \Sigma_c$.

Let $\Sigma = \Sigma_1 \cup \Sigma_2$, $L_1 \subseteq \Sigma_1^*$, and $L_2 \subseteq \Sigma_2^*$. For i = 1, 2, $s \in \Sigma^*$, and $\sigma \in \Sigma$, we define the natural projection $P_i : \Sigma^* \to \Sigma_i^*$ according to:

$$P_i(\in) = \in, \quad P_i(\sigma) = \begin{cases} \in \text{ if } \sigma \notin \Sigma_i \\ \sigma \text{ if } \in \notin \Sigma_i \end{cases}, \; P_i(s\sigma) = P_i(s)P_i(\sigma)$$

The map $P_i^{-1} : Pwr(\Sigma_i^*) \to Pwr\Sigma^*$ is the inverse image of P_i such that for $L \subseteq \Sigma_1^*$, $P_i^{-1}L : \{s \in \Sigma^* | P_i(s) \in L\}$

Definition 1. For $\mathbf{G}_i = (Q_i, \Sigma_i, \delta_i, q_{o,i}, Q_{m,i})$ $(i = 1,2)$, we define the synchronous product $G = \mathbf{G}_1 \| \mathbf{G}_2$ of the two DES as:

$$G := (Q_1 \times Q_2, \Sigma_1 \cup \Sigma_2, \delta, (q_o,1, q_o,2), Q_{m,1} \times Q_{m,2})$$

where $\delta((q_1, q_2), \sigma)$ is only defined and equals

(q_1', q_2') if $\sigma \in (\Sigma_1 \cap \Sigma_2), \delta_1(q_1, \sigma) = q_1', \delta_2(q_2, \sigma) = q_2'$ or

(q_1', q_2) if $\sigma \in \Sigma_1 - \Sigma_2, \delta_1(q_1, \sigma) = q_1'$ or

(q_1, q_2') if $\sigma \in \Sigma_2 - \Sigma_1, \delta_2(q_2, \sigma) = q_2'$.

Citation: Mulahuwaish A, Alsuwaidan A, Leduc RJ. Timed fault-tolerant supervisory control. *Int Rob Auto J.* 2024;10(2):48–61.
DOI: 10.15406/iratj.2024.10.00283

It follows that $L(\mathbf{G}) = P_1^{-1}L(\mathbf{G}_1) \cap P_2^{-1}L(\mathbf{G}_2)$ and $L_m(\mathbf{G}) = P_1^{-1}L_m(\mathbf{G}_1) \cap P_2^{-1}L_m(\mathbf{G}_2)$. We note that if $\Sigma_1 = \Sigma_2$, we get $L(\mathbf{G}) = L(\mathbf{G}_1) \cap L(\mathbf{G}_2)$ and $L_m(\mathbf{G}) = L_m(\mathbf{G}_1) \cap L_m(\mathbf{G}_2)$

For DES, the two main properties we want to check are nonblocking and controllability.

Definition 2. A DES **G** is said to be nonblocking if

$$\overline{L_m(\mathbf{G})} = L(\mathbf{G})$$

Definition 3. Supervisor **S** is controllable with respect to plant **G** if for all $s \in L(\mathbf{S}) \cap L(\mathbf{G})$,

$$\text{Elig}_{L(\mathbf{S})}(s) \supseteq \begin{cases} \text{Elig}_{L(\mathbf{G})}(s) \cap (\Sigma_u \cup \{\tau\}) & \text{if } \text{Elig}_{L(\mathbf{S}) \cap L(\mathbf{G})}(s) \cap \Sigma_{for} = \emptyset \\ \text{Elig}_{L(\mathbf{G})}(s) \cap \Sigma_u & \text{if } \text{Elig}_{L(\mathbf{S}) \cap L(\mathbf{G})}(s) \cap \Sigma_{for} \neq \emptyset \end{cases}$$

TDES properties

For TDES, we have the addition properties of activity loop free and proper timed behavior. The first definition ensures that the clock tick cannot be delayed indefinitely, while the second ensures that either a tick or an untimed event (which cannot be disabled) is always possible in the plant.

Definition 4. TDES $\mathbf{G} = (Q, \Sigma, \delta, q_o, Q_m)$ is activity-loop-free (ALF) if

$$(\forall q \in Q_r)(\forall s \in \Sigma_{act}^*)\delta(q, s) \neq q$$

Definition 5. A plant TDES **G** has proper time behavior if:

$$(\forall q \in Q_r)(\exists \sigma \in \Sigma_u \cup \tau)\delta(q, \sigma)!$$

Fault-tolerant setting

In this section, we will introduce our concept of fault events, a consistency property that our systems must satisfy, and the four fault scenarios that we want our supervisors to be able to handle. Our eventual goal will be to be able to determine if our supervisor will be controllable for our plant in a given fault scenario. In the following section, we will assume that all DES are deterministic, and that we are given plant $\mathbf{G} = (Y, \Sigma, \delta, y_o, Y_m)$ and supervisor $\mathbf{S} = (X, \Sigma, \xi, x_o, X_m)$

Fault events

In this paper, our approach will be to add a set of uncontrollable events to our plant model to represent the possible faults in the system. Our goal will be to design supervisors that will still behave correctly (i.e. stay controllable and nonblocking) when a fault event occurs, even though they can't detect the fault event directly. We start by defining a group of $m \geq 0$ mutually exclusive sets of fault events.

$$\Sigma_{Fi} \subseteq \Sigma_u,\ i = 1, \ldots, m$$

The idea here is to group related faults into sets such that faults of a given set represent a common fault situation, while faults of a different set represent a different fault situation. Consider our illustrative example from Section 1.1, specifically the track loop shown in Figure 2. It would make sense to group the fault events for sensors 9 and 10 as they could both be used to detect a train before it enters the next track segment. However, a fault event for sensor 16 would not be relevant for this task so we would put it into a different fault set.

Definition 6. We refer to faults in Σ_{Fi}, $i = 1, \ldots, m$, collectively as standard fault events:

$$\Sigma_F := \dot{\bigcup}_{i=1,\ldots,m} \Sigma_{Fi}$$

We note that for $m = 0$, $\Sigma_F = \emptyset$.

The standard fault events are the faults that will be used to define the various fault scenarios that our supervisors will need to be able to handle. However, there are two additional types of faults that we need to define in order to handle two special cases. The first type is called unrestricted fault events, denoted $\Sigma_{\Omega F} \subseteq \Sigma_u$. These are faults that a supervisor can always handle and thus are allowed to occur unrestricted. For our example in Section 1.1, this might be a fault associated with a sensor that is not used at all by the system's supervisor and could thus be safely ignored.

The second type is called excluded fault events, denoted $\Sigma_{\Delta F} \subseteq \Sigma_u$. These are faults that cannot be handled at all and thus are essentially removed in our scenarios. The idea is that this would allow us to still design a fault-tolerant supervisory for the remaining faults.

From our example in Section 1.1, consider sensor 13 from Figure 2. If we wished to stop a train at this sensor so it could be loaded by a crane, we would be unable to do so if the sensor failed as there is not a second sensor located close enough to stop the train at the correct location. If we modelled a fault at this sensor, we would have to make it an excluded fault or the system would fail all fault-tolerant tests. This is an example of a fault that could not be handled by a supervisor, and would need to be addressed by adding an additional backup sensor to the system.

For each fault set, Σ_{Fi}, $i = 1, \ldots, m$, we also need to define a matching set of reset events, denoted $\Sigma_{Ti} \subseteq \Sigma$. These events will be explained in Section 3.3, when we describe the resettable fault scenario.

Timed fault-tolerant consistency

We now present a consistency requirement that our timed system must satisfy, the timed fault-tolerant (TFT) consistency definition. This is an extension of the faulttolerant (FT) consistency definition from Mulahuwaish,[1,3,4] where the only difference is that Point 7 is new. It thus follows that if a system is TFT consistent it is also FT consistent. We note that as the tick event is controllable, Definition 7 implies that tick cannot be a fault event.

Definition 7. A system, with a plant $\mathbf{G} = (Y, \Sigma, \delta, y_o, Y_m)$, a supervisor $\mathbf{S} = (X, \Sigma, \xi, x_o, X_m)$ and fault and reset sets Σ_{Fi}, Σ_{Ti} $(i = 1, .., m)$, $\Sigma_{\Delta F}$, and $\Sigma_{\Omega F}$ is timed fault-tolerant (TFT) consistent if:

(1) $\Sigma_{\Delta F} \cup \Sigma_{\Omega F} \cup \Sigma_F \subseteq \Sigma_u$

(2) $\Sigma_{\Delta F}, \Sigma_{\Omega F}, \Sigma_{Fi}$ $(i = 0, .., m)$, are pair-wise disjoint.

(3) $(\forall i \in 1, .., m)\Sigma_{Fi} \neq \emptyset$

(4) $(\forall i \in 1, .., m)\Sigma_{Fi} \cap \Sigma_{Ti} = \emptyset$

(5) Supervisor **S** is deterministic.

(6) $(\forall x \in X)(\forall \sigma \in (\Sigma_{\Omega F} \cup \Sigma_{\Delta F} \cup \Sigma_F))\ \xi(x, \sigma) = x$

(7) $(\Sigma_{\Delta F} \cup \Sigma_{\Omega F} \cup \Sigma_F) \cap \Sigma_{for} = \emptyset$

Point (1) says that fault events are uncontrollable since allowing a supervisor to disable fault events would be unrealistic. **Point (2)** requires that the indicated sets of faults be disjoint since they must each be handled differently. **Point (3)** says that fault sets Σ_{Fi} are non-empty. **Point (4)** says a fault set must be disjoint from its corresponding set of reset events so we can distinguish them.

Points (5) and **(6)** say that **S** is deterministic and that at every state in **S**, there is a selfloop for each fault event in the system. This

means a supervisor cannot change state (and thus change enablement information) based on a fault event. This is a key concept as it effectively makes fault events unobservable to supervisors. If $\mathbf{S}$ is defined over a subset $\Sigma' \subset \Sigma$ instead, we could equivalently require that Σ' contain no fault events.

Point (7) says that there are no forcible, fault events. This is because it would be unrealistic to be able to make a fault event occur on command.

We note that the above definition implies that we do not need to make use of the observability property,[36] saving us the cost of verifying it. Essentially, the observability property is used to check if a partial observation supervisor (one that can only see a subset of the available events) exists that will provide the same closed-loop behavior as an existing supervisor, who can observe all events. As our approach is a verification method that assumes we are given a supervisor that is already forced by the fault-tolerant consistency definition to treat fault events as effectively unobservable (it can't change state based on them), there is no need to verify the observability property as our existing supervisor is already sufficient for our needs.

Fault scenarios

When faults are added to a plant model, we typically can have strings containing so many faults in a row that any controllability or nonblocking test would fail. However, we are typically only interested in knowing if a system will be controllable and nonblocking if only a certain pattern of faults have occurred. For example, we might only want to know if at most one fault occurs, will our system be controllable and nonblocking? Our fault scenarios are an attempt to characterize common fault situations that we would want our supervisors to handle.

In this paper, we will use five faults scenarios that were presented in Mulahuwaish et al.,[1–4] as they are still applicable in the TDES setting. The scenarios range from simple situations easy to verify, to ones that are more flexible in terms of how faults can occur and how often, but more expensive to verify. They are by no means exhaustive, but we felt that they represented a good characterization of situations that would likely be of interest.

The first is the default fault scenario where the supervisor must be able to handle any non-excluded fault event that occurs. The second scenario is the $N \geq 0$ fault scenario where the supervisor is only required to handle at most N non-excluded fault events and all unrestricted fault events. Consider our illustrative example from Section 1.1, specifically the track loop shown in Figure 2. If we wished to prevent a collision in the track segment bounded by sensors 11 and 13, we could stop the train at sensors 9 or 10. We could handle $N = 1$ faults (i.e. sensor 9 or 10 failed but not both), but we could not handle $N = 2$ faults (both sensors failed at the same time).

The next scenario is the one-repeatable fault scenario where the supervisor is only required to handle at most one non-excluded fault event and all unrestricted fault events. This is similar to the N fault scenario with $N = 1$, except that once a given fault has occurred, it can continue to occur, but no other standard fault events may occur.

Consider our illustrative example from Section 1.1, specifically the track loop shown in Figure 2. Applying this scenario, we could for example have a fault occur at sensor 10, but once that occurs we could no longer have faults at sensors 9 and 16, but could continue to have faults at sensor 10. Rather than focusing on how many fault events occurred, the one-repeatable fault scenario focuses on how many components fail. It essentially says at most one component in the system can have a fault, but doesn't restrict how often the component exhibits this fault.

The next scenario is the m-one-repeatable fault scenario where the supervisor is required to handle all unrestricted fault events, but no more than one fault event from any given Σ_{Fi} $(i = 0, \ldots, m)$ fault set, but those events can occur multiple times. This definition allows the designer to group faults together in fault sets such that a fault occurring from one set does not affect a supervisor's ability to handle a fault from a different set.

This scenario extends the one-repeatable fault scenario to allow at most one component to fail per system area associated with a given fault set. If we assume the fault sets from the example in Section 3.1, then this scenario would allow multiple faults to occur at sensors 10 and 16 as they are from separate fault sets, but once a fault occurs at sensor 10, we could no longer get faults at sensor 9 as it is from the same fault set. The last scenario we consider is the resettable fault scenario. This is designed to capture the situation where at most one fault event from each Σ_{Fi} $(i = 1, \ldots, m)$ fault set can be handled by the supervisor during each pass through a part of the system, but this ability resets for the next pass. For this to work, we need to be able to detect when the current pass has completed and it is safe for another fault event from the same fault set to occur. We use the fault set's corresponding set of reset events to achieve this. The idea is that once a reset event has occurred, the current pass can be considered over and it is safe for another fault event to occur.

If we continue the above example, we could have sensors 9 and 10 in one fault set, and set the corresponding reset event set to only contain the detection event for sensor 11. If we get a fault event from sensor 9 and 10 in a row, we would be unable to stop the train. However, if we got a fault from sensor 10 only and then the detection event for sensor 11, we would know we could now safely get a second fault event from sensor 9 or 10 (but not both) and still be able to stop the train. Such a supervisor could handle an infinite number of faults from sensors 9 and 10, as long as they don't both fail during the same pass.

Timed fault-tolerant controllability definitions

In this section, we introduce new timed fault-tolerant controllability definitions so that we can verify if our TDES supervisor will stay controllable for the fault scenarios that we introduced in the previous section. In essence, these definitions characterize strings that belong to the desired fault scenario, and only require supervisors to satisfy the controllability definitions for these strings.

We note that we don't need to introduce corresponding timed fault-tolerant nonblocking definitions, as the ones from Mulahuwaish,[1–4] still apply. This is because the nonblocking property is the same for both the timed and untimed setting. It is also important that the tick event can't be a fault event as this ensures that the nonblocking fault-tolerant properties do not have conflicting definitions.

Due to space limitations, we will only present results for the default, one-repeatable and m-one-repeatable fault scenarios. Please refer to Alsuwaidan[23] for timed properties, algorithms, and correctness proofs for the $N \geq 0$ and resettable fault scenarios.

Timed fault-tolerant controllability

The first fault-tolerant property that we present is designed to handle the default fault scenario. First, we need to define the language of excluded faults. This is the set of all strings that include at least one fault from $\Sigma_{\Delta F}$.

Citation: Mulahuwaish A, Alsuwaidan A, Leduc RJ. Timed fault-tolerant supervisory control. *Int Rob Auto J.* 2024;10(2):48–61.
DOI: 10.15406/iratj.2024.10.00283

Definition 8. We define the language of excluded faults as:

$$L_{\Delta F} = \Sigma^*.\Sigma_{\Delta F}.\Sigma^*$$

Definition 9. A system, with a plant $\mathbf{G} = (Y, \Sigma, \delta, y_o, Y_m)$, a supervisor $\mathbf{S} = (X, \Sigma, \xi, x_o, X_m)$ and fault sets $\Sigma_{Fi}\ (i = 1, \ldots, m)$ and $\Sigma_{\Delta F}$, is timed fault-tolerant (T-FT) controllable if it is TFT consistent and:

$$(\forall s \in L(\mathbf{S}) \cap L(\mathbf{G}))(s \notin L_{\Delta F}) \Rightarrow$$

$$\text{Elig}_{L(\mathbf{S})}(s) \supseteq \begin{cases} \text{Elig}_{L(\mathbf{G})}(s) \cap (\Sigma_u \cup \{\tau\}) & \text{if } \text{Elig}_{L(\mathbf{S}) \cap L(\mathbf{G})}(s) \cap \Sigma_{for} = \emptyset \\ \text{Elig}_{L(\mathbf{G})}(s) \cap \Sigma_u & \text{if } \text{Elig}_{L(\mathbf{S}) \cap L(\mathbf{G})}(s) \cap \Sigma_{for} \neq \emptyset \end{cases}$$

For brevity, when it clear to which fault sets we are referring, we can state this property more concisely as **S** is timed fault-tolerant controllable for **G**.

The above definition is essentially the standard timed controllability definition but ignores strings that include excluded fault events. We note that if $\Sigma_{\Delta F} = \emptyset$, then Definition 9 reduces to the standard controllability definition.

Timed one-repeatable fault-tolerant controllability

The next fault-tolerant property that we introduce is designed to handle the onerepeatable fault scenario. First, we need to define the language of one-repeatable fault events. This is the set of strings that contain at most one fault event from Σ_F, but that event can occur multiple times in the string.

Definition 10. We define the language of one-repeatable fault events as:

$$L_{1RF} = (\Sigma - \Sigma_F)^* \cup \bigcup_{\sigma \in \Sigma_F} \left((\Sigma - \Sigma F)^* . \sigma . (\Sigma - (\Sigma F - \{\sigma\}))* \right)$$

Definition 11. A system, with a plant $\mathbf{G} = (Y, \Sigma, \delta, y_o, Y_m)$, a supervisor $\mathbf{S} = (X, \Sigma, \xi, x_o, X_m)$ and fault sets $\Sigma_{Fi}\ (i = 1, \ldots, m)$ and $\Sigma_{\Delta F}$ is timed one repeatable fault-tolerant (T-1-R-FT) controllable if it is TFT consistent and:

$$(\forall s \in L(\mathbf{S}) \cap L(\mathbf{G}))(s \notin L_{\Delta F}) \wedge (s \in L_{1RF}) \Rightarrow$$

$$\text{Elig}_{L(\mathbf{S})}(s) \supseteq \begin{cases} \text{Elig}_{L(\mathbf{G})}(s) \cap (\Sigma_u \cup \{\tau\}) & \text{if } \text{Elig}_{L(\mathbf{S}) \cap L(\mathbf{G})}(s) \cap \Sigma_{for} = \emptyset \\ \text{Elig}_{L(\mathbf{G})}(s) \cap \Sigma_u & \text{if } \text{Elig}_{L(\mathbf{S}) \cap L(\mathbf{G})}(s) \cap \Sigma_{for} \neq \emptyset \end{cases}$$

The above definition is essentially the standard timed controllability definition, but ignores strings that include excluded fault events, and strings that contain more than two unique fault events from Σ_F. We note that if $m = 0$ we get $\Sigma_F = \emptyset$. This means Definition 11 simplifies to the TFT controllable definition.

Timed m-one-repeatable fault-tolerant controllability

The next fault-tolerant property that we introduce is designed to handle the m-one repeatable fault scenario. First, we need to define the language of m-one-repeatable fault events. This is the set of all strings that contain at most one fault event from a given fault set $\Sigma_{Fi}\ (i = 1, \ldots, m)$, but that event can occur multiple times in the string. We note that a string in L_{1RF_m} could potentially contain a unique event from each different fault set, but no two unique events from the same fault set.

Definition 12. We define the language of m-one-repeatable fault events as:

$$L_{1RF_m} = \bigcap_{i=1}^{m} (\Sigma - \Sigma F_i)* \cup \bigcap_{\sigma \in \Sigma_{Fi}} (\Sigma - \Sigma F_i)^* . \sigma . (\Sigma - (\Sigma F_i - \{\sigma\}))*)$$

Definition 13. A system, with plant $\mathbf{G} = (Y, \Sigma, \delta, y_o, Y_m)$, supervisor $\mathbf{S} = (X, \Sigma, \xi, x_o, X_m)$ and fault sets $\Sigma_{Fi}\ (i = 1, \ldots, m)$ and $\Sigma_{\Delta F}$ is timed m-one-repeatable fault tolerant (T-m-1-R-FT) controllable, if it is TFT consistent and:

$$(\forall s \in L(\mathbf{S}) \cap L(\mathbf{G}))(s \notin L_{\Delta F}) \wedge (s \in L1RF_m) \Rightarrow$$

$$\text{Elig}_{L(\mathbf{S})}(s) \supseteq \begin{cases} \text{Elig}_{L(\mathbf{G})}(s) \cap (\Sigma_u \cup \{\tau\}) & \text{if } \text{Elig}_{L(\mathbf{S}) \cap L(\mathbf{G})}(s) \cap \Sigma_{for} = \emptyset \\ \text{Elig}_{L(\mathbf{G})}(s) \cap \Sigma_u & \text{if } \text{Elig}_{L(\mathbf{S}) \cap L(\mathbf{G})}(s) \cap \Sigma_{for} \neq \emptyset \end{cases}$$

The above definition is essentially the standard timed controllability definition, but ignores strings that include excluded fault events, and strings that contain more than one unique fault event from the same fault set. We note that if $m = 0$ we get $\Sigma_F = \emptyset$. This means Definition 13 simplifies to the TFT controllable definition.

Algorithms

In this section, we will present algorithms to construct and verify the timed faulttolerant controllability properties that we defined in Section 4. We will not present an algorithm for the TFT consistency property as its individual points can easily be checked by adapting various standard algorithms.

We assume that the our TDES system consists of a plant $\mathbf{G} = (Y, \Sigma, \delta, y_o, Y_m)$, supervisor $\mathbf{S} = (X, \Sigma, \xi, x_o, X_m)$ and fault and reset sets $\Sigma_{Fi}, \Sigma_{Ti}\ (i = 1, \ldots, m), \Sigma_{\Delta F}$, and $\Sigma_{\Omega F}$. We also assume that the timed controllability and synchronous product algorithms are given. We use *vTCont* (**Plant, Sup**) to indicate timed controllability verification, and to indicate timed controllability verification, and || to indicate the synchronous product operation.

Similar to the untimed fault-tolerant algorithms in Mulahuwaish,[1–4] our approach will be to construct plant components to synchronize with our plant G such that the new TDES will restrict the occurrence of faults to match the given timed fault-tolerant controllability definitions. We can then synchronize the plant components together and then use a standard controllability algorithm to check the property. This approach allows us to automatically take advantage of existing scalability methods such as incremental[24] and binary decision diagram-based (BDD) algorithms.[25–30]

Since every TDES must contain the tick event, we add a tick event selflooped at every state in the plants we construct. Moreover, all the constructed plants have all of their states marked so that we do not directly change the system's marked behavior.

Algorithms to construct plants

We will now discuss the algorithms required to construct the needed plant components for the various timed fault-tolerant algorithms. This will require the construction of two different types of plants. Figures 7 and 8 show examples of these plants. We will not discuss the plant component needed to verify the timed one-repeatable fault-tolerant properties as it is essentially a special case of the timed m-one-repeatable fault-tolerant plant component. Please refer to Mulahuwaish[1] for details.

Figure 7 Timed excluded faults plant $G_{tΔF}$.

Citation: Mulahuwaish A, Alsuwaidan A, Leduc RJ. Timed fault-tolerant supervisory control. *Int Rob Auto J.* 2024;10(2):48–61.
DOI: 10.15406/iratj.2024.10.00283

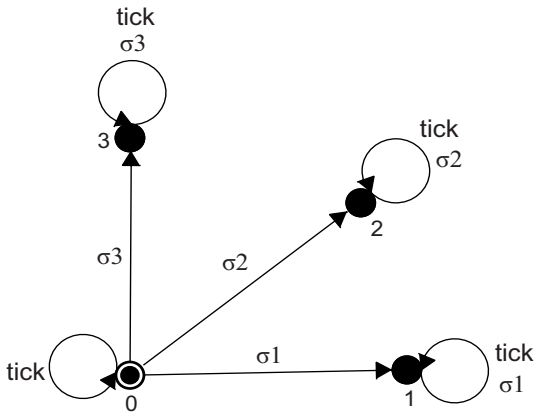

Figure 8 Timed m-One-Repeatable fault plant $G_{t1RF,i}, \Sigma_{F_i} = \{\sigma_1, \ldots, \sigma_3\}$.

Figure 7 shows an example timed excluded faults plant, $G_{t\Delta F}$. This is a TDES with event set $\Sigma_{\Delta F} \cup \{\tau\}$ one selflooped transition for tick, and a marked, initial state. This will have the effect of removing any $\Sigma_{\Delta F}$ transitions from any DES it is synchronized with. The algorithm to construct $G_{t\Delta F}$ is given by Algorithm 1.

Algorithm 1 construct- $G_{t\Delta F}$

1: $Y_1 \leftarrow \{y_0\}$

2: $Y_{m,1} \leftarrow Y_1$

3: $\delta_1 \leftarrow \delta_1 \cup \{(y_0, \tau, y_0)\}$

4: return $(Y_1, \Sigma_{\Delta F} \cup \{\tau\}, \delta_1, y_o, Y_{m,1})$

Figure 8 shows an example timed m-one-repeatable faults plant, $\mathbf{G_{t1RF,i}}$ $(i \in \{1, \ldots, m\})$. This is a TDES with event set $\Sigma_{Fi} \cup \{\tau\}$, and $k+1$ marked states, where $k = |\Sigma_{Fi}|$. It has a transition for each fault event in Σ_{Fi} from the initial state to a new state unique to that fault event. There is also a selflooped transition at that state for that event. Moreover, it creates one selflooped transition for tick at each state. Synchronizing with this TDES will allow at most on unique fault event from Σ_{Fi} to occur, but that event can occur multiple times. The algorithm to construct $\mathbf{G_{t1RF,i}}$ is given by Algorithm 2.

Algorithm 2 construct- $\mathbf{G_{t1RF,i}}$

1: $k \leftarrow |\Sigma_{F_i}|$

2: $Y_1 \leftarrow \{y_0, \ldots, y_k\}$

3: $Y_{m,1} \leftarrow Y_1$

4: $\delta_1 \leftarrow \emptyset$

5: $j \leftarrow 1$

6: $\delta_1 \leftarrow \delta_1 \cup \{(y_0, \tau, y_0), (y_j, \tau, y_j)\}$

7: for $\sigma \in \Sigma_{F_i}$

8: $\delta_1 \leftarrow \delta_1 \cup \{(y_0, \sigma, y_j), (y_j, \sigma, y_j)\}$

9: $j \leftarrow j + 1$

10: end for

11: return $(Y_1, \Sigma_{F_i} \cup \{\tau\}, \delta_1, y_o, Y_{m,1})$

Verify timed fault-tolerant controllability

We will now discuss the algorithms to verify our timed fault-tolerant controllability properties. We will not discuss the algorithms to verify the timed one-repeatable fault-tolerant controllability property as they are essentially a special case $m = 1$ of the timed m-one-repeatable fault-tolerant controllability property. Please refer to Mulahuwaish[1] for details. $m = 1$

Algorithm 3 shows how to verify timed fault-tolerant controllability for $\mathbf{G}$ and S. TDES $G_{t\Delta F}$ contains the excluded fault events but no transitions except for a *tick* selfloop at the initial state, synchronizing with $G_{t\Delta F}$ will remove all the excluded fault transitions, but allow *tick* transitions to occur without restriction. Checking that S is controllable for the resulting behavior will have the effect of verifying timed fault-tolerant controllability.

Algorithm 3 Verify timed fault-tolerant controllability

1: $G_{t\Delta F} \leftarrow construct - G_{t\Delta F}(\Sigma_{\Delta F})$

2: $\mathbf{G'} \leftarrow \mathbf{G} \,||\, G_{t\Delta F}$

3: pass$\leftarrow$vTCont$(\mathbf{G'}, S)$

4: return pass

Algorithm 4 shows how to verify timed m-one-repeatable fault-tolerant controllability for $\mathbf{G}$ and S. As $G_{t\Delta F}$ removes any excluded fault transitions, and each $\mathbf{G_{t1RF,i}}$ allows at most one unique fault event but that event can occur multiple times, checking that S is controllable for the resulting behavior will have the effect of verifying timed m-one-repeatable fault-tolerant controllability.

Algorithm 4 Verify timed m-one-repeatable fault-tolerant controllability

1: $G_{t\Delta F} \leftarrow construct - G_{t\Delta F}(\Sigma_{\Delta F})$

2: for $i = 1, \ldots, m$

3: $\mathbf{G_{t1RF,i}} \leftarrow construct - \mathbf{G_{t1RF,i}}(\Sigma_{F_i}, i)$

4: end for

5: $\mathbf{G'} \leftarrow \mathbf{G} \| G_{t\Delta F} \| \mathbf{G_{t1RF,1}} \| \ldots \| \mathbf{G_{t1RF,m}}$

6: pass $\leftarrow$ vTCont$(\mathbf{G'}, S)$

7: return pass

Algorithm complexity analysis

In this section, we provide a complexity analysis for the timed fault-tolerant controllability algorithms. In the following subsections, we assume that our system consists of a plant $G = (Y, \Sigma, \delta, y_o, Y_m)$, supervisor $\mathbf{S} = (X, \Sigma, \xi, x_o, X_m)$, and fault and reset sets Σ_{Fi}, Σ_{Ti} $(i = 1, \ldots, m)$, $\Sigma_{\ddot{A}F}$, and $\Sigma_{\Omega F}$.

In this paper, we will base our analysis on the complexity analysis from Cassandras *et al.,*[22] that states that the untimed controllability algorithms have a complexity of $O(|\Sigma||Y||X|)$, where $|\Sigma|$ is the size of the system event set, $|Y|$ is the size of the plant state set, and $|X|$ is the size of the supervisor state set. In the analysis that follows, $|Y_{t\Delta F}|$ is the size of the state set for $Y_{t\Delta F}$ (constructed by Algorithm 1).

Examining untimed and timed controllability algorithms, (see Rudie[15] and Alsuwaidan[23]) it's easy to see they differ in the constant number of operations they each perform per transition that leaves

Citation: Mulahuwaish A, Alsuwaidan A, Leduc RJ. Timed fault-tolerant supervisory control. *Int Rob Auto J.* 2024;10(2):48–61.
DOI: 10.15406/iratj.2024.10.00283

each reachable state of the closed-loop system. As such, timed controllability also has complexity $O(|\Sigma||Y|\,|X|)$.

Timed FT controllability algorithm

In Algorithm 3, we replace our plant DES by $G' \leftarrow G \,||\, G_{t\Delta F}$. This gives us a worst case state space of $|Y\,||Y_{t\Delta F}|$ for G'. Substituting this into our base algorithm's complexity for the size of our plant's state set gives $O(|\Sigma||Y\,||Y_{t\Delta F}\,||X|)$. As $|Y_{t\Delta F}|=1$ by Algorithm 1, it follows that our complexity is $O(|\Sigma||Y|\,|X|)$ which is the same as our base algorithm.

Timed one-repeatable FT controllability algorithm

The complexity of the timed one-repeatable FT controllability algorithm can be obtained from the analysis of the timed m-one-repeatable FT controllability algorithm by taking $N_F = |\Sigma_F|$ and $m=1$. It thus follows that verifying timed one-repeatable FT controllability increases the complexity of verifying controllability by a factor of $|\Sigma_F|+1$.

Timed m-one-repeatable FT controllability algorithm

For Algorithm 4, we replace our plant DES by $G' = G\|G_{t\Delta F}\|G_{t1RF,1}\|\ldots\|G_{t1RF,m}$. This gives us a worst case state space of $|Y||Y_{t\Delta F}||Y_{t1RF,1}|\ldots|Y_{t1RF,m}|$ for G', where $|Y_{t1RF,i}|$ is the size of the state set for $G_{t1RF,i}$ $(i=1,\ldots,m)$, which is constructed by Algorithm 2. Substituting this into our base algorithm's complexity gives

$$O(|\Sigma||Y||Y_{t\Delta F}\,||Y_{t1RF,1}|\ldots|Y_{t1RF,m}|\,|X|).$$

We note that $|Y_{t\Delta F}|=1$ by Algorithm 1, and $|Y_{t1RF,i}|=|\Sigma_{Fi}|+1$ $(i=1,\ldots,m)$ by Algorithm 2. If we take N_F as an upper bound of all $|\Sigma_{Fi}|$, we get $O((NF\ 1)m|\Sigma||Y|\,|X|)$. It thus follows that verifying timed m-one-repeatable FT controllability increases the complexity of verifying controllability by a factor of $(N_F+1)^m$.

Algorithm correctness

In this section, we introduce several propositions and theorems that show that the algorithms introduced in Section 5 correctly verify that a TFT consistent system satisfies the corresponding timed fault-tolerant controllability properties from Section 4.

Timed fault-tolerant propositions

The propositions in this section will be used to support the timed fault-tolerant controllability theorems in Section 6.2. Timed fault-tolerant controllability definitions are essentially controllability definitions with the added restriction that a string s is only tested if it is satisfies the appropriate timed fault-tolerant property from Section 4.

The timed fault-tolerant controllability verification algorithms are intended to replace the original plant with a new plant G', such that G' is restricted to strings with the desired property. Propositions 1-2 essentially assert that strings belongs to the closed behaviour of G', if and only ifs satisfies the appropriate timed fault-tolerant controllable property from Section 4 (i.e. the string belongs to the desired scenario).

The first proposition asserts that strings belongs to the closed behaviour of G', if and only ifs satisfies the needed pre-requisite for the timed fault-tolerant controllable property.

Proposition 1 Let system with supervisor $S = (X, \Sigma, \xi, x_o, X_m)$ and plant $G = (Y, \Sigma, \delta, y_o, Y_m)$ be TFT consistent, and let G' be the plant constructed in Algorithm 3. Then:

$$(\forall s \in L(G))\, s \notin L_{\Delta F} \Leftrightarrow s \in L(G')$$

Proof. **See Appendix**.

The next proposition asserts that string S belongs to the closed behaviour of G', if and only if S satisfies the needed pre-requisite for the timed m-one-repeatable fault-tolerant controllable property.

Proposition 2 Let system with supervisor $S = (X, \Sigma, \eta, x_o, X_m)$ and plant $G = (Y, \Sigma, \delta, y_o, Y_m)$ be TFT consistent, and let G' be the plant constructed in Algorithm 4. Then:

$$(\forall s \in L(G))(s \notin L_{\Delta F}) \wedge (s \in L_{1RFm}) \Leftrightarrow s \in L(G')$$

Proof. **See Appendix**.

Timed fault-tolerant controllable theorems

In this section we present theorems that show the timed fault-tolerant controllable algorithms in Section 5 will return *true* if and only if the timed fault-tolerant consistent system satisfies the corresponding timed fault-tolerant controllability property. Due to space limitations, we will not present results for the timed one-repeatable fault-tolerant controllability and nonblocking properties as they can be handled as a special case ($m=1$) of the timed m-one-repeatable fault-tolerant properties. Please refer to Mulahuwaish[3] for details.

Theorem 1 states that verifying that our system is timed fault-tolerant controllable is equivalent to verifying that our supervisor is controllable for the plant G' constructed by Algorithm 3. We will only give the proof for Theorem 2 as it is very similar, but more complicated.

Theorem 1

Let system with supervisor $S = (X, \Sigma, \xi, x_o, X_m)$ and plant $G = (Y, \Sigma, \delta, y_o, Y_m)$ be TFT consistent, and let G' be the plant constructed in Algorithm 3. Then S is *timed fault-tolerant controllable* for **G** iff S is controllable for G'.

Proof. See Alsuwaidan.[23]

Theorem 2 states that verifying that our system is timed m-one-repeatable fault-tolerant controllable is equivalent to verifying that our supervisor is controllable for the plant G' constructed by Algorithm 4.

Theorem 2 Let system with supervisor $S = (X, \Sigma, \xi, x_o, X_m)$ and plant $G = (Y, \Sigma, \delta, y_o, Y_m)$ be TFT consistent, and let G' be the plant constructed in Algorithm 4. Then S is timed m-one repeatable fault-tolerant controllable for G iff S is controllable for G'.

Proof. **See Appendix**.

Manufacturing example

This example is based on the small example from Mulahuwaish,[2,3] which in turn was based on the system described in Leduc.[32] The testbed was designed to simulate a manufacturing workcell using model train equipment, in particular problems of routing and collision. We will discuss a single-loop version of the example, as shown in Figure 2. This example consists of eight sensors and two trains (*train 1*, *train 2*). Train 1 starts between sensors 9 and 10, while train 2 starts between sensors 15 and 16. Both trains can only traverse the tracks in a counter-clockwise direction.

This example builds upon the illustrative example that we introduced in Section 1.1, providing the remaining plant models for the example, as well as the details of how we applied our timed fault-

tolerant approach to the example. We recommend that you reread Section 1.1 to refresh your memory of the details presented there, as they will not be repeated below.

Plant models

The plant models, for the portion of the testbed we are currently considering, consists of the following basic elements: sensors, trains and the relationship between sensors and trains.

Sensor Models

In Section 1.1, we introduced the eight TDES plant models for our eight sensors. We first presented the original sensor models (without fault events added) in Figure 3. We then presented new models, for sensors $J \in \{9,10,16\}$, with the added fault events. For this example, we will use the original models for sensors $J \in \{11,\ldots,15\}$, and the new models for sensors $J \in \{9,10,16\}$ as we are assuming that only these sensors have faults. This restriction is done to simplify the example and make it easier to illustrate our approach.

We now need to define our fault and reset event sets for the example. We set $\Sigma_{\Delta F} = \Sigma_{\Omega F} = \emptyset$ as our example does not require any fault events of this type. We also set $m = 4$, $\Sigma_{F1} = \{t1F\_at9, t1F\_at10\}$, $\Sigma_{F2} = \{t1F\_at16\}$, $\Sigma_{F3} = \{t2F\_at9, t2F\_at10\}$, $\Sigma_{F4} = \{t2F\_at16\}$. We group our fault events in this manner as sensors 9 and 10 are both relevant to preventing a train from entering the track segment delineated by sensors 11 and 13, while sensor 16 is not. Also, the faults in detecting one train, are not relevant to the faults in detecting the other train, for our example.

Finally, we define our corresponding reset event sets as follows: $\Sigma_{T1} = \{t1\_at11\}$, $\Sigma_{T2} = \{t1\_at14\}$, $\Sigma_{T3} = \{t2\_at11\}$, and $\Sigma_{T4} = \{t2\_at14\}$. These are chosen as they represent the given train reaching a section of track past the sensors associated with the given fault set.

Train models

The train models are shown in Figure 9 for train K $(K = 1,2)$. Train K can only move when its enablement event *en_trainK* occurs, and then it can move at most a single unit of distance (event *umv_trainK*), before another *en_trainK* must occur. This allows a supervisor to precisely control the movement of the train by enabling and disabling event *en_trainK* as needed.

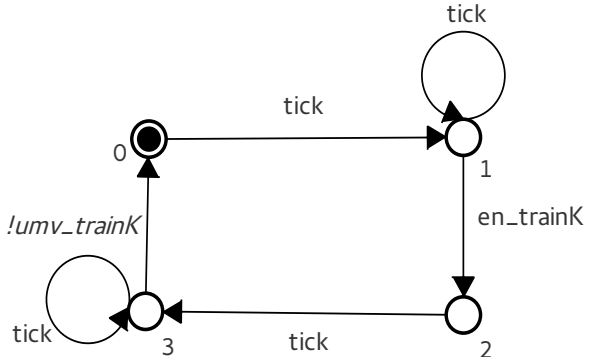

Figure 9 Train K $(K = 1,2)$ with tick events.

Relationship between sensors and trains models

Figure 10 shows the relationship between train K's $(K = 1,2)$ movement, and a sensor detecting the train. It captures the idea that a train can reach at most one sensor during a unit movement, and no sensors if it is disabled, also Figure 10 shows the replacement model, one for each train, with fault events added. We now seen that our plant model contains 14 DES in total.

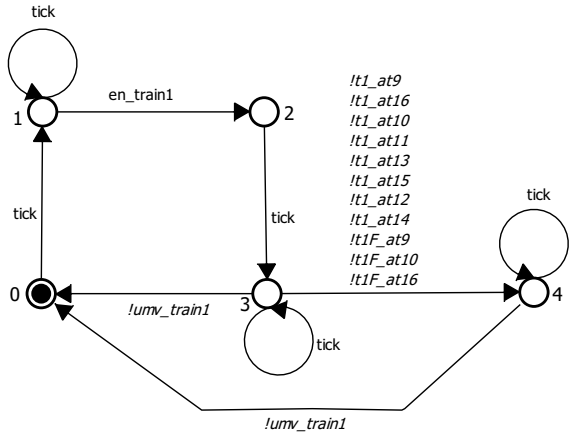

Figure 10 Sensors and Train K $(K = 1,2)$ with fault and tick events.

Adding forcing

To extend Alsuwaidan's example, we have added forcing for events *en_trainK* $(K = 1,2)$. However, this is not straightforward to do in a modular way as these events are not always possible in the plant. Also, multiple supervisors will need to enable and force these events. If a supervisor tries to force the event when either it isn't possible in the plant or disabled by another supervisor, the result could be uncontrollable.

To handle this problem, we have introduced two new controllable events forceT1 and forceT2, shown in Figures 11 and 12. Now, the collision protection supervisors in Section 7.2 will disable these events instead of *en_trainK* events, to signal when the train is allowed to move or not. We note that as these events are added as part of the supervisor's implementation, they are assumed to occur very quickly after they are enabled.

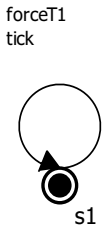

Figure 11 Add forceT1 event.

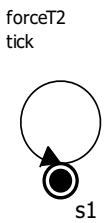

Figure 12 Add forceT2 event.

We now need to add supervisors to force the *en_trainK* events to occur right away, as long as they are eligible and not disabled. This is accomplished by the *doForceTK* supervisors, shown in Figures 13 and 14. These supervisors handle the forcing by first waiting until the *en_trainK* event is possible in the plant, and then waiting for the forceTK event to occur. Once forceTK occurs, the tick event is disabled until the *en_trainK* event has occurred, forcing the event. The

forceTK event is required to coordinate with the collision protection supervisors so that doForceTk doesn't try to force the *en_trainK* event when it has been disabled, which would have caused the supervisor to be uncontrollable.

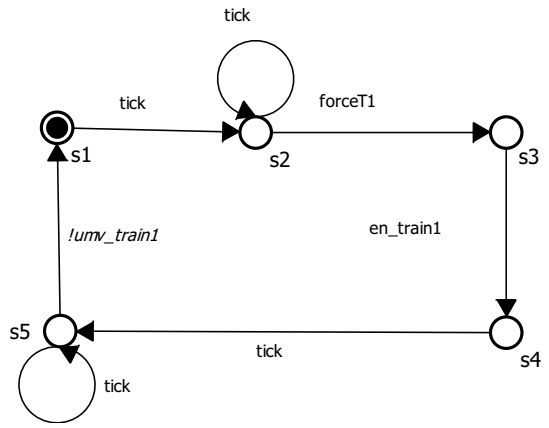

Figure 13 Force_en train1 for train 1.

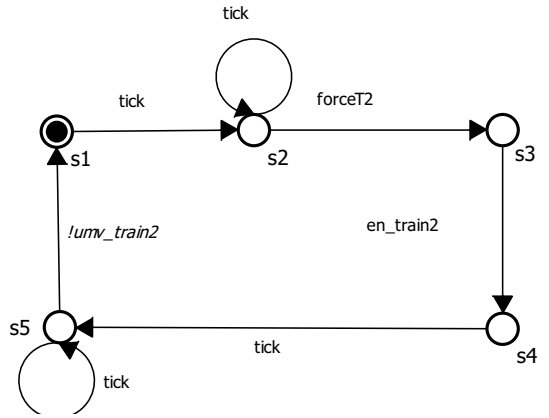

Figure 14 Force_en train2 for train 2.

Modular supervisors

After the plant models were developed, four supervisors were designed to prevent collisions in the track sections with sensors 11-13, 15-16, 12-14, and 9-10. The idea is to ensure that only one train uses this track section at a time.

Below we present two versions of the collision protection supervisors. The first version is based upon the original collision protection supervisors from Leduc[11] which were designed with the assumption that the system did not contain faults. The second version is a new fault-tolerant version with added redundancy.

Collision protection supervisors

Figure 15 shows the fault-tolerant collision protection supervisor **(CPS-11-13FT)** for the track section containing sensors 11 and 13. The original version **(CPS-11-13)** is identical except that the *t1_at9* and the *t2_at9* transitions are not present. Once a train has reached sensor 11, the other train is stopped at sensor 10 until the first train reaches sensor 15, which indicates it has left the protected area. The stopped train is then allowed to continue. Figures 17, and show similar fault-tolerant supervisors for two of the remaining track sections. Again, the original version is identical except that the *t1_at9* and the *t2_at9* transitions are not present.

Figure 16, shows the final collision protection supervisor. It is unchanged as it does not depend on the sensors with faults. We also note that supervisors **CPS-15-16** and **CPS-9-10** have nonstandard initial states in order to reflect the starting locations of the two trains.

It's easy to see that the original supervisor **CPS-11-13** will not be fault-tolerant as it relies solely on sensor 10 to detect when a second train arrives. If sensor 10 fails, the train continues and could collide with the first train. Supervisors CPS-9-10 and **CPS-12-14** will also not be fault-tolerant because of sensor 10. A failure at sensor 10 could cause supervisor **CPS-9-10** to miss a train entering the protected zone, and could cause supervisor **CPS-12-14** to miss a train leaving the protected zone.

Fault-tolerant collision protection supervisors

We next modified supervisor **CPS-11-13** to make it more fault-tolerant. The result is shown in Figure 15. We have added at states 1 and 4 a check for both sensor 9 or sensor 10. That way if sensor 10 fails but sensor 9 doesn't, we can still stop the train at sensor 9 and avoid the collision. We made similar changes to supervisors **CPS-12-14**, and **CPS-9-10**, as shown in Figures 17, and 18.

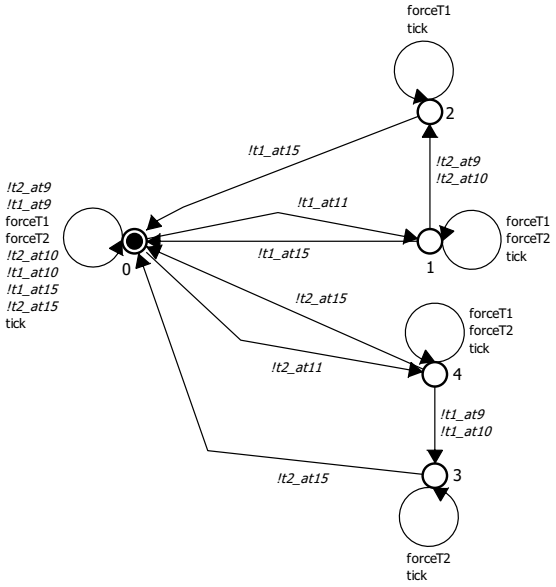

Figure 15 CPS-11-13FT supervisor.

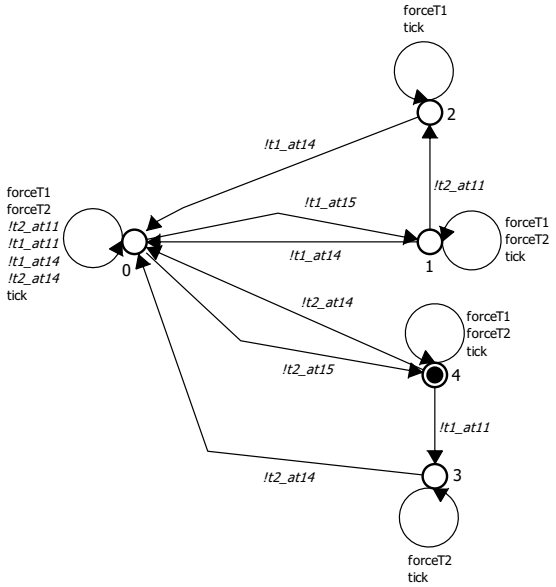

Figure 16 CPS15-16 supervisor.

Citation: Mulahuwaish A, Alsuwaidan A, Leduc RJ. Timed fault-tolerant supervisory control. *Int Rob Auto J.* 2024;10(2):48–61.
DOI: 10.15406/iratj.2024.10.00283

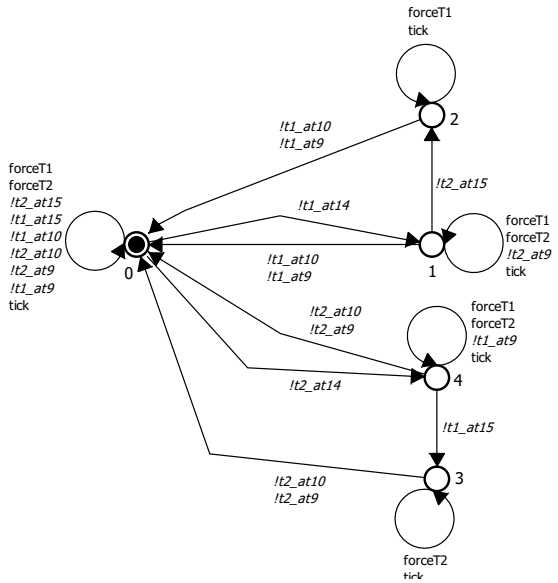

Figure 17 CPS-12-14FT supervisor.

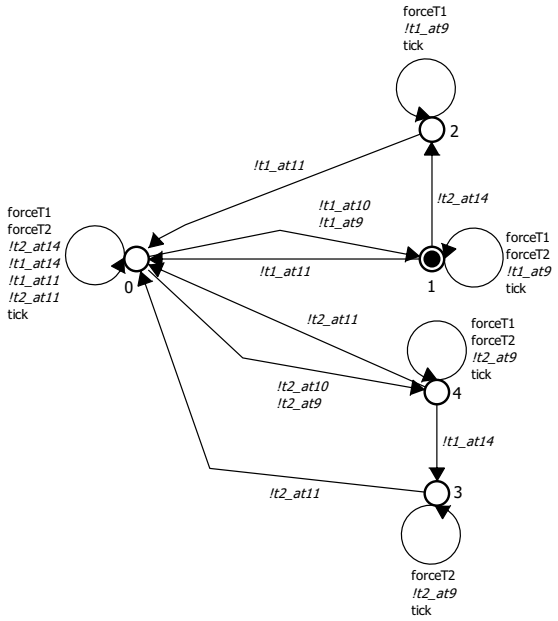

Figure 18 CPS-9-10FT supervisor.

Discussion of results

Using our software research tool, DESpot,[32] we were able to determine that the system is timed one-repeatable FT controllable, and timed m-one-repeatable FT controllable. We also not that the system failed the FT controllable and nonblocking properties as expected, since they would allow the fault events to occur unrestricted. Table 1 shows the test results, system state sizes, and runtime for these tests. Runtime data is from DESpot's binary decision diagram-based (BDD)[25–30] algorithms as the timed m-one-repeatable fault-tolerant statesize was too large for the automata-based algorithms.

Table 1 Example results

		Verification Time (seconds)			
Property	**State Size**	**Timed Cont.**		**Nonblocking**	
Timed fault-tolerant	10,502,000	0	F	0	F
Timed one-repeatable fault-tolerant	49,933,600	0	P	I	P
Timed m-one-repeatable fault-tolerant	271,118,000	0	P	I	P

We next examine the state sizes of each evaluated system, which includes the plant components added as part of the verification process. We first note that for our example, we have $m = 4$, $|\Sigma_F| = 35$, $N_F = 2$, and a base-system state size of 10,502,000. From Section 5.3, we expect that (worst case) the one-repeatable FT and m-one-repeatable FT algorithms would multiply our base-system state size by factors of $|\Sigma_F| + 1 = 36$ and $(N_F + 1)^m = 3^4 = 81$, respectively.

Examining Table 1, we see that these two algorithms actually contribute multiplying factors of 4.75, and 25.82, respectively. We see immediately that the actual increase in complexity for this example is much less than expected, in particular for the one-repeatable FT algorithms.

Conclusions and future work

In this paper we investigated the problem of fault-tolerance (FT) for timed discreteevent systems. We extended the existing fault-tolerant supervisory control approach of Mulahuwaish[1–4] to include timing information. We introduced our setting and providing different fault scenarios. We then provide three timed fault-tolerant definitions to verify that the system will remain controllable in each scenario.

This approach is different from the typical fault-tolerant methodology as the approach does not rely on detecting faults and switching to a new supervisor; it requires a supervisor to work correctly under normal and fault conditions. This is a passive approach that relies upon inherent redundancy in the system being controlled.

Our approach provides an easy method for users to add fault events to a system model and is based on user designed supervisors and verification. As synthesis algorithms have higher complexity than verification algorithms, our approach should be applicable to larger systems than existing active fault-recovery methods that are synthesis based. Also, modular supervisors are typically easier to understand and implement than the results of synthesis.

Finally, our approach does not require expensive (in terms of algorithm complexity) fault diagnosers to work. Diagnosers are, however, required by existing methods to know when to switch to a recovery supervisor. As a result, the response time of diagnosers is not an issue for us. Our supervisors are designed to handle the original and the faulted system. However, the tradeoff is that our approach may result in an overly cautious supervisor.

We then present a set of algorithms to verify timed controllability for each scenario. We then proved that the algorithms correctly evaluated

Citation: Mulahuwaish A, Alsuwaidan A, Leduc RJ. Timed fault-tolerant supervisory control. *Int Rob Auto J.* 2024;10(2):48–61.
DOI: 10.15406/iratj.2024.10.00283

the timed fault-tolerant controllability properties that we introduced. They can instantly take advantage of existing controllability and nonblocking software, as well as scalability approaches such as incremental verification and binary decision diagrams (BDD).

We then present a set of algorithms to verify the timed fault-tolerant properties. As these algorithms involve adding new plant components and then checking standard timed controllability, they can instantly take advantage of existing controllability software, as well as scalability approaches such as incremental verification and binary decision diagrams (BDD).

For each algorithm, we provide a complexity analysis showing that the TFT algorithms multiply the complexity of the standard algorithms by a factor of (1), $\left(\left|\Sigma_F\right|+1\right)$, and $(N_F+1)^m$ where m is the number of fault sets, $\left|\Sigma_F\right|$ is the number of fault events, and N_F is an upper bound of all $\left|\Sigma_{Fi}\right|$ $\left(i = 1, \ldots, m\right)$. We then prove the correctness of the algorithms.

We finish with a small manufacturing example that illustrates how the theory can be applied.

For future work, it would be useful to extend a timed fault-tolerant method to the sampled-data setting[32] in order to address concurrency and implementation issues. We would also like to extend the approach to the hierarchical interface-based supervisory control (HISC).[33–36] The information hiding and encapsulation properties of HISC should allow us to scale our approach up to handle much larger systems.

Acknowledgments

None.

Conflicts of interest

Authors declare that there is no conflict of interest.

References

1. Mulahuwaish A. *Fault-tolerant supervisory control*. PhD. thesis, Department of Computing and Software, McMaster University, 2019.
2. Mulahuwaish A, Leduc RJ. Fault-tolerant supervisory control with permanent faults. *Int J Control*. 2020;96(4):823–839.
3. Mulahuwaish A, Radel S, Dierikx O, et al. Fault tolerant supervisory control. *IFAC-PapersOnLine.* 2021;48(7):124–131.
4. Radel S, Mulahuwaish A, Leduc R. *Fault tolerant controllability*. American control conference, Chicago: USA. 2015
5. Ramadge P, Wonham WM. Supervisory control of a class of discrete-event processes. *SIAM J Control Optim.* 1987;25(1):206–230.
6. Wonham WM, Cai K. Supervisory Control of Discrete-Event Systems. Springer; 2019.
7. Wonham, WM, Ramadge P. On the supremal controllable sublanguage of given language. *SIAM J Control Optim*. 1987;25(3):637–659
8. Brandin B, Wonham W. The supervisory control of timed discrete-event systems. in decision and control. In: Proceedings of the 31st IEEE Conference on. 1992;4:3357–3362.
9. Brandin B, Wonham WM. Supervisory control of timed discrete-event systems. *IFAC-PapersOnLine.* 1994;39(2):329–342.
10. Brandin BA. Real-time supervisory control of automated manufacturing systems. Ph.D. thesis, Department of Electrical Engineering, University of Toronto. 1993. Also appears as Systems Control Group technical report # 9302, Department of Electrical Engineering, University of Toronto.
11. Leduc R. PLC implementation of a DES supervisor for a manufacturing testbed: an implementation perspective. Master's thesis, Dept. of Elec and Comp Eng, University of Toronto, Toronto, Ont; 1996.
12. Bourdon S, Lawford M, Wonham W. Robust nonblocking supervisory control of discrete-event systems. *IEEE Transactions on Automatic Control.* 2005;50(12):2015–2021.
13. Lin F. Robust and adaptive supervisory control of discrete event systems. *IEEE Trans. Automatic Control.* 1993;38(12):1848–1852.
14. Saboori A, Zad SH. Robust nonblocking supervisory control of discrete-event systems under partial observation. *Systems & Control Letters*. 2006;55(10):839–848.
15. Rudie K. Software for the control of discrete-event systems: A complexity study. Master's thesis, Dept. of Electrical and Computer Engineering, University of Toronto, Toronto, Ont; 1988.
16. Sampath M. Sengupta R, Lafortune S, et al. Failure diagnosis using discrete-event models. *IEEE Trans Control System Technology*. 1996;4(2):105–124.
17. Wen Q, Kumar R, Huang J, et al. A framework for fault-tolerant control of discrete event systems. *IEEE Trans on Automatic Control*. 2008;53(8);1839–1849.
18. Paoli A, Sartini M, Lafortune S. Active fault tolerant control of discrete event systems using online diagnostics. *Automatica* 2011;47(4):639–649.
19. Park SJ, Lim JT. Fault-tolerant robust supervisor for discrete event systems with model uncertainty and its application to a workcell. *IEEE Transactions on Robotics and Automation*. 1999;15(2):386–391.
20. Allahham A, Alla H. Monitoring of timed discrete events systems with interrupts. automation science and engineering. *IEEE Transactions.* 2010;7(1):146–150.
21. Moosaei M, Zad S. *Modular fault recovery in timed discrete-event systems: application to a manufacturing cell*. In: Proceedings of 2005 IEEE Conference on Control Applications. 2005:928–933.
22. Cassandras C, Lafortune S. *Introduction to discrete event systems.* 2nd edn. Springer. 2009.
23. Alsuwaidan A. *Timed fault tolerant supervisory control*. Master's thesis; Dept. of computing and software, McMaster University. 2016.
24. Brandin BA, Malik R, Malik P. Incremental verification and synthesis of discrete-event systems guided by counter-examples. *IEEE Trans. on Control Systems Technology*. 2004;12(3):387–401.
25. Bryant AE. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys*. 1992;24:293–318.
26. Ma C. *Nonblocking supervisory control of state tree structures*. PhD. thesis, Department of Electrical and Computer Engineering, University of Toronto. 2004.
27. Song R, Ryan J Ledu C. Symbolic synthesis and verification of hierarchical interface-based supervisory control. Master's thesis, Dept. of Comput. and Softw., McMaster University, Hamilton, Ont; 2006.
28. Vahidi A, Lennartson B, Fabian M. Efficient analysis of large discrete-event systems with binary decision diagrams. In: Proc. of the 44th IEEE Conf. Decision Contr. and and 2005 European Contr. Conf. 2005;2751–2756.
29. Wang YB. Eng. Sampled-data supervisory control. Master's thesis, Dept. of Computing and Software, McMaster University, Hamilton, Ont; 2009.
30. Zhang Z. Smart TCT: an efficient algorithm for supervisory control design. Master's thesis, Dept. of Electrical and Computer Engineering, University of Toronto, Toronto, Ont; 2001.
31. Leduc RJ, Wang Y, Ahmed F. Sampled-data supervisory control. *Discrete Event Dynamic Systems*. 2014;24(4):541–579.
32. DESpot: DESpot project. 2013.

33. Leduc RJ. Hierarchical interface-based supervisory control with data events. *International Journal of Control*. 2009;82(5):783–800.

34. Leduc RJ, Brandin BA, Lawford M, et al. Hierarchical interface-based supervisory control, part I: Serial case. *IEEE Trans. Automatic Control*. 2005;50(9):1322–1335.

35. Leduc RJ, Lawford M, Dai P. Hierarchical interface-based supervisory control of a flexible manufacturing system. *IEEE Trans. on Control Systems Technology*. 2006;14(4):654–668.

36. Leduc RJ, Lawford M, Wonham WM. Hierarchical interface-based supervisory control, part II: Parallel case. *IEEE Trans. Automatic Control.* 2005;50(9):1336–1348.

37. Lin F, Wonham W. On observability of discete-event systems. *Inform Sci.* 1988;44(3):173–198.

Citation: Mulahuwaish A, Alsuwaidan A, Leduc RJ. Timed fault-tolerant supervisory control. *Int Rob Auto J.* 2024;10(2):48–61.
DOI: 10.15406/iratj.2024.10.00283