Reliability Guarantees in Automata Based Scheduling for Embedded Control Software

Santhosh Prabhu, Aritra Hazra, Pallab Dasgupta
Department of CSE, IIT Kharagpur
West Bengal, India - 721302.
Email: {santhosh.prabhu, aritrah, pallab}@cse.iitkgp.ernet.in

Abstract—Automata based scheduling is a recent technique for online scheduling of software control components in embedded systems. This paper studies one important aspect of automata based scheduling that has not been studied in the past, namely resilience to faults. The goal of the proposed technique is to create an automaton that recommends the scheduling patterns that are admissible with respect to control performance requirements, when the state of the system has been mutated by faults. The problem has been formulated as a game between the scheduler and the (possibly faulty) system, where a winning strategy of the scheduler prevents the system from reaching bad states forever. We present a method for analyzing the structure of the game and extracting an automaton that captures the winning strategies of the scheduler, namely the automaton to be used for automata based scheduling.

Index Terms—Embedded Control Systems, Automata-based Scheduling, Fault-tolerant System, Reliable Scheduling.

I. INTRODUCTION

A software controlled embedded system typically consists of a control software that executes on a computational platform and interacts with an analog environment (called the plant) through the sensors and actuators of the system. The control software consists of a set of control actions of the controller components that are scheduled periodically to perform a desired activity and achieve the required performance for the overall system. Each component has to be scheduled according to some pattern so that the control meets performance requirements like exponential stability.

Traditionally, the software components are executed according to pre-determined hard periodic schedules. This typically results in poor utilization of the computational resources. Research performed in the last decade established that the patterns of scheduling that are admissible with respect to some of the important control performance requirements can be symbolically described by $\omega$-regular languages [5]. Researchers also showed [1], that it is possible to compute a finite state automaton representing all acceptable schedules of a control component. The product of such automata for the components that run on the same processor yields an automaton representing the set of schedules that are globally admissible, that is, they satisfy the control performance requirement of all components. Every random walk in this automaton represents a valid global schedule, and therefore this automaton can be used to choose schedules on-the-fly during the runtime of the controller. This emerging approach is known as automata based scheduling.

An important aspect of automata based scheduling, that has been overlooked in the past, is in exploring the ability of automata based scheduling to adapt to faults in the system. For example, delay in arrival of the message carrying the sensor input may cause a software component to execute with stale data. In spite of the best intentions of the scheduler, the control did not apply as required, and it may become necessary to rearrange the schedule over the next few cycles to prevent control performance from being affected. Since automata based scheduling proposes the departure from the traditional approach of adhering to pre-computed schedules, it opens up the possibility of online resilience to faults by such dynamic rearrangement of the scheduling pattern in the immediate future of the occurrence of the fault. The goal of this paper is to study this aspect of automata based scheduling.

The automaton created by the approach of [1], [5] does not guarantee control performance under non-ideal situations. A scheduler that intends to guarantee control performance under non-ideal situations must refrain from schedules that are potentially risky, that is, ones in which a fault can drive the system to a state from which no schedule can avoid violation of control performance requirements. In other words, we need an automaton that generates schedules with the more stringent guarantee that control performance will not be compromised even in the presence of a fault.

The main contribution of this paper is to prove that when control performance is determined by exponential stability, it is possible to construct an automaton that can choose schedules on-the-fly that are robust to $k$ number of failures within a period of $L$, where $k$ and $L$ are given parameters. We formulate the problem as a game between the scheduler (the protagonist) and the non-ideal system (the antagonist). The protagonist wins if at any (finite) time, the control performance falls below acceptable standards. The protagonist wins if this never happens, that is, the protagonist guarantees to honor the exponential stability requirement at all times. We present a method for finding a winning strategy (if it exists) for the protagonist that works with constant memory. We also show that the set of winning strategies of the protagonist can be captured by a finite state automaton, which can then be used in automata based robust scheduling.
The paper is organized as follows. Section II presents the background relating scheduling with stability criteria of embedded controllers. The formal problem statement is given in Section III. Section IV describes the game-theoretic formulation of the problem and Section V presents the automata-based approach to generate reliable control schedules. Section VI concludes this paper.

II. BACKGROUND

Formally, the dynamics of the physical plant can be approximated by a given discrete-time linear time-invariant system, described as:

\[
\begin{align*}
    x_p(t + 1) &= A_p x_p(t) + B_p u(t) \\
    y(t) &= C_p x_p(t)
\end{align*}
\]

The state of the plant is defined by the set of plant variables, \( x_p \), the output of the plant is \( y \), and the control input of the plant is \( u \). The matrices \( A_p \), \( B_p \), and \( C_p \) denote the state transition matrix, the input map, and the output map for the plant model. Equation (1) defines the state transition function of the plant, and Equation (2) defines the plant output.

The feedback control software reads the plant output, \( y \), and adjusts the control variables, \( u \). The controller can be designed as a linear time-invariant system such that the stability \([2], [3]\) of the whole system (the controller and the plant together) is guaranteed when the output of the plant is fed to the input of the controller and vice versa.

Formally, it is assumed that the controller of such an embedded system can be represented as a discrete-time linear time-invariant system:

\[
\begin{align*}
    x_c(t + 1) &= A_c x_c(t) + B_c u(t) \\
    y(t) &= C_c x_c(t)
\end{align*}
\]

where, \( x_c \) is the state of the controller and \( A_c, B_c \) and \( C_c \) are, respectively, the state transition matrix, the input map, and the output map for the controller. Equation (3) and Equation (4) define the controller state transition function and the output function, respectively. The dynamics of the composed system (the controller and the plant together) can be described by transformations of the form:

\[
x(t + 1) = \begin{pmatrix} A_p & B_p C_p \\ B_c & A_c \end{pmatrix} x(t)
\]

where \( x = (x_p^T, x_c^T)^T \) is the concatenation of the states of the plant and the controller.

We can represent the sequence of control actions applied on the environment by one or more controllers, by a sequence of transformation matrices, \( A_{\sigma_1}, A_{\sigma_2}, \ldots \), where each \( A_{\sigma} \) is chosen from an alphabet \( \Sigma \) of transformation matrices. The matrix in Equation 5 is one such transformation matrix. These matrices may be interpreted in different ways, depending on the nature of the controller in use.

(i) In the case of a single controller capable of operating in different modes, each transformation matrix may correspond to the transformation it applies in a particular mode of operation, as described in [5].

(ii) When multiple controllers operate simultaneously, a transformation matrix represents the combined transformation applied on the environment, by the set of controllers active at a particular instant.

(iii) In the case of multiple software controllers that share an ECU, with no two controllers executing simultaneously, each transformation matrix corresponds to the transformation applied when a particular controller executes. We assume that the transformation applied by the remaining controllers (which are inactive) can be represented by an identity transformation matrix.

We assume that we have multiple control components, one or more of which may at the same time (the second scenario). But the methodology presented is generic, and can be used in any of the above scenarios. We denote by \( C_{\sigma} \), a particular combination of controllers that may execute together. A word \( \sigma = \sigma_1 \sigma_2 \ldots \) (where each \( \sigma_i \in [1, m] \)) specifies a schedule, which denotes the sequence of combinations \( C_{\sigma} = C_{\sigma_1} C_{\sigma_2} \ldots \). The sequence of transformations corresponding to this schedule is given by \( A_{\sigma} = A_{\sigma_1} A_{\sigma_2} \ldots \).

One of the standard control requirements is that of exponential stability. A system (whose state is defined in terms of \( n \) variables) is said to be \((L, \epsilon)\)-exponentially stable, given the parameters \( L \in \mathbb{N} \) and \( \epsilon \in (0, 1] \), if \( ||x(t + L)||/||x(t)|| < \epsilon \) for every \( t \in \mathbb{N} \). It follows from control theory and the work presented by Weiss and Alur [1], [5] that the exponential stability requirement can be captured by the following language:

\[
\text{ExpStab}(L, \epsilon) = \{ \sigma \in \Sigma^* : ||A_{\sigma_{t+1}} \cdots A_{\sigma_{t+L}}|| < \epsilon \text{ for every } t \in \mathbb{N} \}
\]

This definition means that an infinite sequence, \( Z \), of transformations satisfies the exponential stability requirement if \( ||A_{\sigma_{t+1}} \cdots A_{\sigma_{t+L}}|| < \epsilon \) for every \( L \) length subsequence \( A_{\sigma_{t+1}} \cdots A_{\sigma_{t+L}} \) of \( Z \). We shall refer to sequences of transformations satisfying the exponential stability requirement as \( \omega \)-regular. The language can be expressed as:

\[
L(A) = \Sigma^\omega - \Sigma^* \eta \Sigma^\omega
\]

where \( \eta \) consists of the \( L \)-length sequences of transformations that violate the exponential stability criteria. In [1], it was shown that it is possible to construct a finite state automaton, such that any infinite random walk in it represents an admissible sequence of transformations and vice versa.

III. THE FORMAL PROBLEM STATEMENT

The role of the automata-based scheduler described in [1] is to ensure that the sequence of control components that are scheduled adheres to the admissible patterns with respect to the stability criteria. In reality other factors such as delay in receiving sensory inputs and electrical faults affecting the sensors / actuators can lead to non-ideal application of control, even when the software components are scheduled correctly. Following the terminology of the previous section, a fault at
a time step is manifested as a mutation on the transformation scheduled in that time step. In other words, we consider scenarios where the scheduler selected some transformation \( A_i \), but the actual execution resulted in a transformation, \( \hat{A}_i \), which differs from \( A_i \) in terms of the updation of the control variables that are affected by the fault.

In general different types of faults may be manifested as different mutations on a transformation matrix, \( A_i \). Our approach is capable of handling multiple types of faults, but for ease of presentation we consider a single type of fault. Hence, for each transformation, \( A_i \in \Sigma \), there is a single mutation, \( \hat{A}_i \), representing the effective transformation when the fault occurs at the time that \( A_i \) is scheduled. We denote by \( \Sigma_f \) the set of mutated versions of transformations in \( \Sigma \).

Next we define the fault patterns as a function of two parameters, \( L \) and \( k \). \( L \) is the window length used in the definition of the stability requirement, as described in the previous section. \( k \) is a given upper bound on the number of occurrences of the fault within each window of length \( L \).

**Example 1:** Consider a braking system, which reduces the speed of a vehicle using a combination of brakes and throttle control. We have given two transformation matrices \( A_R \) and \( A_S \), corresponding to application of brakes and adjusting the throttle. Assume that a fault may cause the apply brake action to manifest incorrectly at most once \((k = 1)\) in every 3 cycles \((L = 3)\), resulting in the transformation matrix \( \hat{A}_S \) being applied. We are given that the stability requirement admits the following set of sequences:

\[
\{(A_S, A_R, A_S), (A_S, A_R, A_R), (A_S, A_S, A_R),
\hat{A}_S, A_R, A_S), (A_R, A_R, A_S),
(A_R, A_S, A_R), (A_R, A_R, \hat{A}_S)\}
\]

The scheduler must ensure that in every window of three steps, the transformation applied is confined to these sequences – even in the presence of faults.

Our problem statement consists of the following two goals:

1. Given an alphabet, \( \Sigma \), of transformation matrices corresponding to the actions scheduled by the scheduler, and an alphabet, \( \Sigma_f \), of the mutated transformation matrices, we seek a scheduling strategy that guarantees exponential stability even if \( k \) or fewer transformations selected by the scheduler in each \( L \) length window are mutated due to fault occurrences.

2. In a given state of the system, there may exist multiple scheduling strategies that achieve the previous objective. Therefore we seek an oracle which returns the set of scheduling alternatives at a given state of the system. This oracle will be based on the automaton that we shall construct.

The following section formulates these problems as a game and presents the solution to the first goal. The second goal is addressed subsequently.

**IV. Game-Based Solution**

Careful examination of the interaction between the scheduler and the environment reveals its competitive flavor, which can be modeled as a game between the two. The scheduler assumes the role of the protagonist, and the environment that of the antagonist. In general, the state of a system can be defined in terms of the history of transformations applied so far, and any scheduling strategy can be defined as a function for selecting a transformation at a given state of the system.

**Definition 1:** The scheduling strategy \( SCHED \) is defined as a function from the sequence of transformations applied so far, to the transformation that will be scheduled next. Mathematically, \( SCHED : (\Sigma \cup \Sigma_f)^* \rightarrow \Sigma \).

Given the sequence, \( \pi \), of transformations scheduled so far, \( SCHED \) determines the transformation \( A_i \) that is to be scheduled next. The sequence produced by scheduling \( A_i \) may be \( \pi \circ A_i \) or \( \pi \circ \hat{A}_i \), depending on whether the antagonist introduces a fault at that state. With this notion, we define a tree \( \tau \) in which each state represents a history of applied transformations, and the two outgoing edges from each state represent possible transformations that may be applied next. The two edges represent the correct and mutated application of the transformation chosen by \( SCHED \). A sequence of transformations is a win for the antagonist (and loss for the protagonist) if it contains a subsequence of length \( L \) that violates the requirement for exponential stability (that is, it contains a subsequence from the set \( \eta \) of Section II). At a given state of the system, the scheduler (protagonist) has a winning strategy if it can guarantee that the actual sequence of transformations does not result in a win for the antagonist.

To determine whether the scheduler has a winning strategy, we define an implicit game graph, \( G = (V, E) \). The state of the game is represented by \( L \)-length sequences over \( \Sigma \cup \Sigma_f \) and the turn of the two players. A move of the protagonist is to schedule a member \( A_i \) of \( \Sigma \) and the following move of the antagonist is to choose between \( A_i \) and \( \hat{A}_i \). Formally:

- \( V = V_S \cup V_E \), is the set of vertices each corresponding to an \( L \)-length sequence. \( V_E \) represents the vertices where the antagonist (environment) gets its turn. \( V_S \) represents the vertices where the protagonist (scheduler) gets its turn. Each vertex represents a state of the game given by the previous \( L \) transformations and is symbolically described by the turn and an \( L \)-length sequence over \( \Sigma \cup \Sigma_f \) where at most \( k \) transformations may be mutated. This restriction ensures that the vertices in the game-tree represent only sequences that satisfy the fault model.

There is an important difference between \( V_E \) and \( V_S \) in terms of the \( L \)-length sequences in their representations. Since the protagonist never selects a transformation from \( \Sigma_f \) in the move preceding the antagonist, the \( L \)-length sequence corresponding to a vertex in \( V_E \) must necessarily end with a transformation from \( \Sigma \). There is no such restriction for the \( L \)-length sequences corresponding to the vertices in \( V_S \) because the antagonist is free to choose from \( \Sigma \) or \( \Sigma_f \).

We shall use the standard array notation, \( v[1 \cdots L] \), to denote the \( L \)-length sequence corresponding to vertex \( v \).

- \( E \subseteq (V_S \times V_E) \cup (V_E \times V_S) \) such that for each \( (u, v) \in E \), the following conditions are satisfied.

  a) If \( u \in V_S \) and \( v \in V_E \), \( u[2 \cdots L] = v[1 \cdots L - 1] \). The transformation chosen by the antagonist at \( u \) becomes \( v[L] \).

  b) If \( u \in V_E \) and \( v \in V_S \), \( u[1 \cdots L - 1] = v[1 \cdots L - 1] \) and \( v[L] \) is either \( A_i \) or \( \hat{A}_i \), where \( A_i \in \Sigma \) is the transformation in \( u[L] \). In other words, the antagonist either mutate the
transformation chosen by the protagonist in the previous move, or leaves it as it is.

A round of the game consists of a move by the protagonist (choosing the next transformation) followed by a move by the antagonist (to mutate or retain the chosen transformation). From the definition of $E$, it follows that in the state $u$ reached at the end of a round, $u[1 \cdots L]$ represents the previous $L$ transformations applied on the system. We assume that our analysis starts beyond the start-up phase of the controller, such that the initial state also has a history of $L$ previous transformations.

The protagonist loses the game at the end of a round, if the state $u \in V_S$ reached at that round is such that the sequence $u[1 \cdots L]$ belongs to the set, $\eta$, of inadmissible sequences for exponential stability (as defined in Section II). Our algorithm starts by labeling these members of $V_S$ as LOSS nodes.

The game graph can be unfolded into a game tree of possibly infinite depth. We shall show that a finite unfolding of the game graph is sufficient to determine whether a winning strategy exists for the protagonist. The finite unfolding is defined as follows:

1. Nodes labeled as LOSS nodes are not expanded any further.
2. A node in $u \in V_S$ in the game tree is marked as a WIN node if a previous occurrence of $u$ appears in the path from the root to the current occurrence of $u$. Nodes labeled as WIN nodes are not expanded any further.

Figure 1 illustrates a part of the game tree for the braking system described in Example 1.

It is easy to see that the above unfolding creates a game tree of finite depth, since no node in $V_S$ is expanded more than once on any path of the game tree. We use the standard Min-Max algorithm [4] to determine whether the game tree has a winning strategy for the protagonist.

**Lemma 1**: If the antagonist has a winning strategy from a node $v$ in the game tree, and there is another occurrence of $v$ on the path from the root, then the antagonist has a winning strategy from the previous occurrence of $v$ that does not involve the latter occurrence of $v$. □

Lemma 1 justifies our approach of labeling the WIN nodes in the game tree. We use the Min-Max algorithm to mark the admissible moves for the protagonist (namely, the moves that do not allow the antagonist to win) at each node $v \in V_S$ of the game tree. We denote by $T$ the tree consisting of these moves and the possible counter-moves by the antagonist to these moves.

**Theorem 1**: The scheduler can guarantee a schedule satisfying the exponential stability criterion if and only if the protagonist has a winning strategy in our game formulation. □

**V. AUTOMATON FOR GENERATING RELIABLE SCHEDULES**

In Section IV we demonstrated that it is possible to construct a finite depth game tree, $T$, representing the problem of finding winning strategies for scheduling. At a given state of the game, the protagonist (scheduler) may have multiple winning strategies offering different moves (transformations) from that state of the game. All of these moves are admissible moves since they belong to some winning strategy of the protagonist.

We construct an automaton where the states correspond to all $L$-length sequences over $\Sigma \cup \Sigma_f$ and transitions correspond to such admissible moves of the protagonist and their mutated alternatives. Formally, our automaton is $A = (Q, \Sigma, q_0, \delta)$ where:

- $Q = V_S$, corresponding to all $L$-length sequences over $\Sigma \cup \Sigma_f$
- $\Sigma$, the set of transformation matrices is the input alphabet
- $q_0$, is a given start-state
- $\delta \subseteq Q \times \Sigma \times Q$ is the transition relation, defined as follows. $(u, A_i, v) \in \delta$ if there is an admissible outgoing transition on protagonist action $A_i$ from state $u$ in $T$, and the state $v$ is such that $u[2 \cdots L] = v[1 \cdots L - 1]$ and either $v[L] = A_i$ or $v[L] = \hat{A}_i$.

In other words, for a state $u \in V_S$ in $T$ having an admissible transition with action $A_i$, our automaton has two non-deterministic transitions – one which adds $A_i$ to the schedule and one which adds $\hat{A}_i$ to the schedule. It may be noted that every path in $T$ has a corresponding run in $A$, and vice versa.

A scheduler uses the automaton $A$ to choose the sequence of transformations. At each state $u$ of the automaton, it can choose a transformation, $A_i$, provided that there exists a transition, $(u, A_i, v)$, in $A$. We define a chosen schedule as the sequence of transformations chosen by the scheduler along some run of $A$.

Our fault model restricts the number of faults in every $L$-length window to at most $k$. Any schedule can be mutated under this restriction to obtain an applied schedule. A schedule is robust with respect to this fault model if and only if the schedule corresponding to the schedule is inadmissible with respect to the exponential stability criteria.

**Theorem 2**: The set of chosen schedules using the automaton $A$ as reference is exactly the set of robust schedules with respect to our fault model. □
VI. CONCLUSION

This paper is the first paper to examine the problem of fault tolerance in automata based scheduling. The main contribution of this paper is to establish that the language of robust schedules with respect to a fault model is $\omega$-regular and can be generated by a finite state automaton. The construction of the automaton relies on a game-theoretic analysis of the scheduling problem. We believe that this approach will open up new vistas for fault tolerant automata based scheduling.

REFERENCES


