Handling Fault Detection Latencies in Automata-based Scheduling for Embedded Control Software

Santhosh Prabhu M, Aritra Hazra, Pallab Dasgupta and P. P. Chakrabarti
Department of Computer Science and Engineering,
Indian Institute of Technology Kharagpur, West Bengal, India - 721302.
Email: {santhosh.prabhu, aritrah, pallab, ppchak}@cse.iitkgp.ernet.in

***Abstract*—There has been recent interest in automata-based scheduling for dynamic adaptation of schedules for embedded control software. Recently we have shown how automata-based scheduling can be extended to account for the possibility of faults in application of control. In this paper, we address the problem of automata-based scheduling when latencies are associated with detection of faults. We show that the game-theoretic approach that is used for handling faults under full visibility may be suitably augmented so as to decide the scheduling strategy in the presence of latencies in fault detection.**

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

I. Introduction

Typically, in any software controlled embedded system, a control software is executed on a computational platform (often on ECUs/processors) and interacts with the *plant* through its sensors and actuators. It enacts a set of periodically scheduled control actions to perform a required activity and achieve the desired performance of the overall embedded system. The control components are scheduled in a way so that it meets some performance requirement like exponential stability.

A typical software controller in an embedded system can have multiple software components that run on shared ECUs. Each of these components is usually responsible for some aspect of the control activity. Generally the task of scheduling the components so as to satisfy the control requirement is done by a scheduler. Every time a component gets scheduled, it controls the system by applying some transformation, which can be represented by a transformation matrix. Hence, scheduling a component effectively means determining the transformation matrix to be applied on the system.

Alur and Wiess [1], [7] have described an automata based solution to the problem of generating schedules that satisfy the *exponential stability* [2], [3], [4] requirement. Their approach is based on the observation that the language of admissible schedules is ω-regular. It involves using a generating automaton to dynamically generate admissible schedules with respect to the stability criterion.

The scheduling problem gets more complicated when there exists a possibility of a fault. A fault causes the chosen control actions of a software component to be applied incorrectly, or not applied at all. When this happens, a *mutated transformation matrix* can be said to be applied on the system. In a previous work [5], we have addressed the problem of automata based scheduling in the presence of faults, and shown that it is possible to construct an automaton for scheduling the components, even under the possibility of faults. The approach formulates the scheduling problem as a game, such that a winning strategy in the game denotes a safe scheduling strategy for the scheduler.

In [5], we have assumed that the scheduler has full visibility of the faults, i.e, the scheduler knows about the fault as soon as it occurs. However, in many real world scenarios, the fault monitoring mechanism might be such, that the occurrence of the fault is not detected immediately by the scheduler. In this paper, we study the problem of scheduling software components in such situations. The primary contributions of this paper is as follows:

- We study the problem of reliable control scheduling where the environment can introduce faults in the applied transformations scheduled by the controller and these faults are not immediately visible to the scheduler.
- We formulate this problem as a partial observability game between environment and scheduler.
- We derive an automaton which can generate schedules that adheres to the exponential stability criterion under partial observability of faults.
- We present a set of experimental results showing the efficacy of our proposed method.

This paper is organized as follows. In Section II, we describe the background for the problem of scheduling software components. The problem is presented formally in Section III. Section IV describes the mechanism for handling latencies in fault detection. We describe the implementation details and present the results in Section VI. Section VII 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:

$$x_p(t+1) = A_p x_p(t) + B_p u(t) \quad (1)$$

$$y(t) = C_p x_p(t) \quad (2)$$

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 denotes 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 [3], [4] 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:

$$x_c(t+1) = A_c x_c(t) + B_c u(t) \quad (3)$$

$$u(t) = C_c x_c(t) \quad (4)$$

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:

$$x(t+1) = \begin{pmatrix} A_p & B_p C_c \\ B_c C_p & A_c \end{pmatrix} x(t) \quad (5)$$

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_{σ_i} is chosen from an alphabet Σ of transformation matrices. The matrix in Equation 5 is one such transformation matrix.

We denote by C_{σ_i} 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, ϵ)-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], [7] that the exponential stability requirement can be captured by the following language:

$$ExpStab(L, \epsilon) = \{\sigma \in I^\omega : \\ ||A_{\sigma_{t+L}} \cdots A_{\sigma_{t+1}}|| < \epsilon \ for \ every \ t \in \mathbb{N}\}$$

This definition means that an infinite sequence, Z, of transformations satisfies the exponential stability requirement iff $||A_{\sigma_{t+L}} \cdots A_{\sigma_{t+1}}|| < \epsilon$ for every L length subsequence $A_{\sigma_{t+1}} \ldots A_{\sigma_{t+L}}$ of Z. We shall refer to sequences of transformations satisfying the exponential stability requirement as *admissible sequences*.

It was shown in [7] that the language of admissible sequences is ω-regular. The language can be expressed as:

$$\mathcal{L}(A) = \Sigma^\omega - \Sigma^* \eta \Sigma^\omega$$

where η 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. Problem Formulation

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, the application of control can be non-ideal due to several other factors such as delay in receiving sensory inputs and electrical faults affecting the sensors / actuators, 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 we consider a single type of fault for ease of presentation. We are given a set Σ of transformation matrices, which correspond to simultaneous control actions by one or more software components. We are also given Σ_f, the mutated versions of the transformation matrices in Σ. We assume that within any given window of length L, a maximum of k faults can occur. Also, it is assumed that the fault is detected by the scheduler after m units of time, where $m \leq L$ (We believe that our approach will hold even when $m > L$, but such scenarios are realistically not possible). Such criteria indicating fault detection latencies introduces a new dimension to the problem formulation and proposed approach in comparison to our previous work [5] where we provided reliable control schedules under faults with full visibility.

However, in this work, our problem is to arrive at a strategy for scheduling the transformation matrices from Σ such that, even in the presence of up to k faults in any window of length L, and the faults being detected only after m time instants, the exponential stability requirement is not violated.

Example 1: Consider a braking system, which reduces the speed of a vehicle using a combination of brakes and throttle control. We are 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. The fault detection latency is assumed to be 1. We are given that the stability requirement admits the following set of sequences:

$$\{\langle A_S, A_R, A_S\rangle, \langle A_S, A_R, A_R\rangle, \langle A_S, A_S, A_R\rangle,$$
$$\langle \hat{A_S}, A_R, A_S\rangle, \langle A_S, A_R, \hat{A_S}\rangle, \langle A_R, A_S, A_S\rangle,$$
$$\langle A_R, A_S, A_R\rangle, \langle A_R, \hat{A_S}, A_R\rangle, \langle A_R, A_R, \hat{A_S}\rangle\}$$

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. □

IV. Scheduling under Faults with Latent Visibility

We have formulated the problem of scheduling with fault detection latencies as a partial observation game between the scheduler, which is the protagonist and the environment, which is the antagonist[1]. In every round of the game, the scheduler decides on a transformation matrix to be scheduled, and the environment chooses whether to mutate it or not, constrained by the assumptions in the fault model. The environment's objective is to force an inadmissible sequence to be scheduled, while the scheduler's is to prevent that from happening. Due to the latency in fault detection, the scheduler is unable to immediately determine which of the possible set of moves the environment has opted for.

The state space for the game is defined by the set of L length sequences over $\Sigma \cup \Sigma_f$, along with the current turn. We define the game graph as $G = (V, E)$ as follows:

- The set V of vertices consists of two types of vertices. Those where the scheduler makes its move (V_S), and those where the environment makes its move (V_E). In addition to the turn, the vertex also stores the L length history of transformation matrices. The sequences in V_S are the ones which are actually applied on the system. On the other hand, the last matrix in the sequences associated with vertices in V_E may get mutated by the environment in its move.
- The set E of edges contains edges between every vertex u to v from V such that,
  1) the players getting the turn in u and v are different
  2) If $u \in V_S$, $seq_u[2 \cdots L] = seq_v[1 \cdots L-1]$, and the edge is labeled by the matrix $seq_v[L]$.
  3) If $u \in V_E$, $seq_u[1 \cdots L-1] = seq_v[1 \cdots L-1]$, the edge is labeled by the matrix $seq_v[L]$, and $seq_v[L] = seq_u[L]$ or $seq_v[L] = \widehat{seq_u[L]}$.
  Here, seq_u and seq_v denote the sequences associated with u and v respectively.

[1]We point out here that the adversarial modeling of the environment has not been made after a qualitative analysis. There may exist non adversarial models of the environment which may also be suitable.

Due to the partially visible nature of the game, the scheduler will not be able to recognize the mutations in the most recent window of length m. The intuitive idea for constructing a winning strategy for the scheduler is quite straightforward – we try to come up with a strategy such that the scheduler's response to every move of environment, will guarantee that the scheduler can force victory irrespective of what move the environment has chosen. The responses to the move begin to differ only when the actual nature of the environment's move becomes known to the scheduler.

To obtain such a strategy for the scheduler, we first expand the graph as a tree and using the Min-max algorithm[6], we annotate each state with win/loss markings. In order to be able to perform such an annotation, it is necessary to first define the winning and losing leaf nodes. The definition of losing leaf nodes is straightforward. Any state whose L length sequence is present in the set of inadmissible sequences is a loss node. For defining the winning leaf nodes for the scheduler, we use the following lemma:

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.

Proof: The antagonist wins if the state of the game reaches a loss node. Therefore, a winning strategy for the antagonist is a subtree, W, of the game tree, T, such that all leaf nodes of W are loss nodes and at each intermediate node of W, the following condition holds: each node $u \in V_E$ in W has a single successor from T in W and each node $u \in V_S$ has all successors from T in W.

Consider the smallest subtree, W, which is a winning strategy from a node v. Then W cannot contain another occurrence of v, otherwise the subtree of W rooted at the second occurrence will be a smaller winning strategy.

Since the subtrees rooted at each occurrence of v are isomorphic, the winning strategies from each occurrence of v are also isomorphic. So, the smallest winning strategy from the first occurrence of v does not involve the subsequent occurrences of v in the game tree. □

On the basis of this lemma, we mark as a winning leaf node any state where the protagonist(the controller) has the turn, and has already been seen in the path to it from the root. Once this is done, the Min-max algorithm may be applied directly. Figure 1 illustrates a portion of the Min-max tree, for the controller described in Example 1. The game is assumed to start from the state corresponding to the sequence $\langle A_S, A_R, A_S\rangle$. If the scheduler chooses A_R in the first round, the environment will be unable to mutate it, and it will be applied correctly. This takes the game to the state corresponding to the sequence $\langle A_R, A_S, A_R\rangle$. Then, if in the second round, the scheduler chooses A_S and the environment chooses not to mutate it, the game reaches the state with sequence $\langle A_S, A_R, A_S\rangle$, which is a repetition, and hence labeled as a winning state. That this tree gives all

possible winning strategies for the controller under the given fault model, when the faults are immediately visible, is stated by the following lemma.

Lemma 2: 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.

Proof (Reproduced from [5]): We recall that a schedule satisfies the exponential stability criterion if the sequence of transformations applied in this schedule does not contain any inadmissible L-length subsequence.

A winning strategy for the protagonist in the game is one that prevents the state of the game to reach any LOSS node, which represents a state of the system where the last L transformations constitute an inadmissible sequence. Therefore a winning strategy for the protagonist guarantees a schedule satisfying exponential stability.

If the protagonist has no winning strategy, then the antagonist can lead the game to a LOSS node. In other words, the antagonist can choose a suitable sequence of mutations for each choice of transformations by the protagonist, such that an inadmissible sequence of transformations is guaranteed to be applied on the system. Now, it remains to be shown that the transformations that the scheduler can choose in order to satisfy exponential stability is no different from the choices available to it in the game as the protagonist, and that the environment as the antagonist in the game is only as powerful as it is in the real world, in terms of the mutations it can introduce.

We make the assumption that any transformation matrix $A_i \in \Sigma$ can be chosen by the protagonist at any instant, irrespective of the history of the game. So, the choices of transformation matrices available to the scheduler in the game is same as the choices available to it for scheduling.

Given that only a maximum of k mutations are possible within any L-length window, and that the mutations are governed by no other constraint, whether the transformation matrix scheduled next can be mutated, is determined entirely by the number of mutations that have been introduced within the last L-length window. This restriction is placed on the antagonist of the game too. So, the antagonist is only as powerful as the environment in the real world. □

Once we obtain the game tree with win/loss markings on each state, we define an equivalence relation R on the set of states. Two states q and q' are said to be equivalent with respect to R if they are indistinguishable for the controller. For example, in Figure 1, the states n_4 and n_5 are equivalent with respect to R, if the fault detection latency is not zero.

We now construct a modified game tree, as follows: For every set $S = \{q_1, q_2 \dots q_n\}$ of states which are equivalent with respect to R, we look for isomorphic winning strategies that are applicable from all $q_i \in S$. All other winning strategies are discarded, by marking the moves as losing moves. If there does not exist such an isomorphic winning strategy, we mark each $q_i \in S$ as a loss node.

After this step, we apply the Min-max algorithm again, and repeat the above procedure. The iteration stops when no more

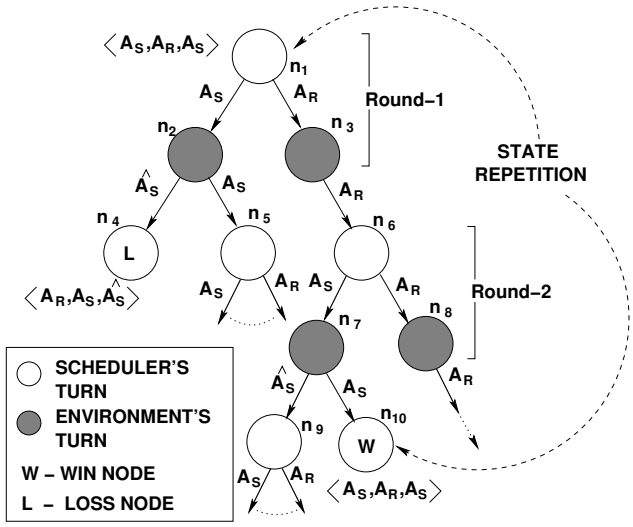

Fig. 1. A section of a Game tree for Example 1

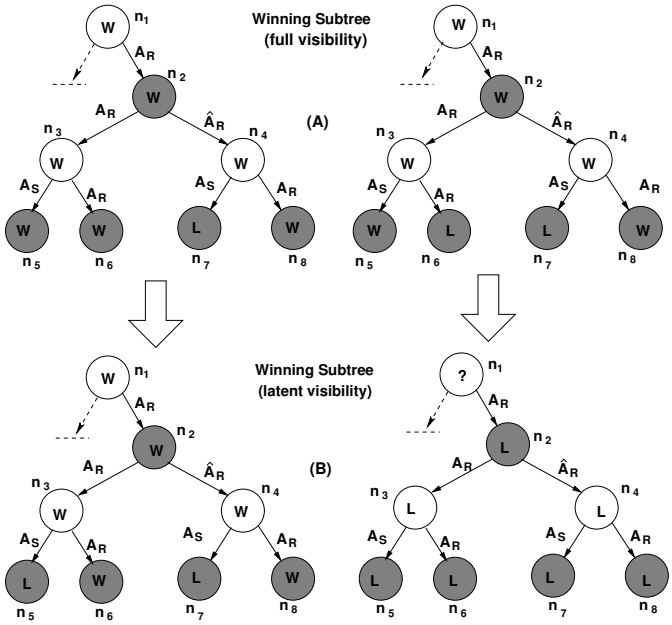

Fig. 2. A snapshot showing the re-labeling of game trees (These trees are not necessarily subtrees of the game tree for the speed governor example).

changes are necessary to the game tree. Figure 2 illustrates how the markings on the game trees may change in each iteration of the above process. In the first game tree, nodes n_3 and n_4 are equivalent, and there exists an isomorphic winning strategy from both nodes, that of choosing A_R. Choosing A_S, however, is not a winning strategy from n_4, and hence, choosing A_S is marked as a losing move from n_3, by relabeling n_5 as a loss node. In the second game tree, the nodes n_3 and n_4 are again equivalent. On applying the same strategy, n_5 and n_8 gets relabeled as loss nodes, which in turn results in n_3, n_4 and n_2 being relabeled as loss nodes. Had there been no latency in fault detection, these nodes would have been winning nodes for the scheduler.

Theorem 1: The re-labeled game tree contains exactly those strategies that do not require the scheduler to behave differently in two states that appear similar to it due to fault detection latencies.

Proof: Since the re-labeled subtree is obtained by dropping strategies from the original tree, and for no two equivalent states is the set of outgoing transitions different, it is clear that every strategy present in the tree is correct.

We now prove that every correct strategy is present in the new tree. Suppose that there exists a subtree for a strategy that was present in the original game tree, but is not present in the new tree. We prove that such a subtree cannot exist.

Consider a subtree of height zero. The root of such a tree will be marked as a losing state, even though it may not be among the inadmissible sequences. This is true because it is not possible for the scheduler to distinguish between this state and some state that is actually a losing state. Such states should be marked as loss, since the attempts by the scheduler to take the game to such a state might actually be leading the game to a losing state.

Now assume that no strategy of upto h rounds is missed. Consider a strategy of $h+1$ rounds. Since the strategy doesn't require the scheduler to behave differently in two states that appear similar, the first move of the strategy must be applicable in all states that are equivalent to the root of the subtree. But if that be the case, the remainder of the strategy would still be in the tree, by our assumption. So, no strategy of $h+1$ rounds is missed.

Thus, by induction, the re-labeled tree contains all strategies in which the scheduler behaves identically on the same information. □

V. Automaton for generating schedules

From the marked Min-max tree for the game formulation, we construct the automaton for generating the schedules. The automaton is defined by a tuple $\langle Q, \Sigma, \delta, q_0 \rangle$, where:

- Q, the set of states of the automaton consists of L length sequences from $(\Sigma \cup \Sigma_f)^k$, in which not more than k symbols are from Σ_f. Also, the last m symbols should necessarily be from Σ.
- Σ is, as defined previously, the set of transformation matrices
- $\delta \subseteq Q \times \Sigma \times Q$ is the transition relation, which is defined as follows. $(u, A_i, v) \in \delta$ if A_i is part of a winning strategy from the state with sequence u where protagonist has the turn and, v satisfies the following conditions:
  1) $v[L] = A_i$ or $v[L] = \hat{A}_i$ and,
  2) v differs at most at one position from u, and if it does, the difference will be that the $L-m+1^{\text{th}}$ symbol in v is the mutated version of the corresponding symbol in u.
- q_0 is the predefined initial sequence

VI. Implementation and Experimental Details

We have implemented an optimized version of the proposed approach, in which we intelligently prune subtrees based on equivalence even as the Min-max algorithm assigns the initial markings to the nodes. For doing this, we keep track of equivalence between the explored states, and whenever the strategy of a state is modified, the strategies of all states equivalent to it are also modified. Whenever a state is found to be losing, the subtrees rooted at that node and all other nodes equivalent to it are pruned.

Table I presents the results obtained from the implementation. The results are obtained for different values of L, different alphabets, different sets of admissible sequences and different values of m and k. Columns 1-3 give the window length (L), maximum number of injected faults (k) and the latency in fault visibility (m), respectively. The number of transformation matrices are reported in Column 4. Column 5 presents the size of the equivalence classes and the required time for the overall analysis (in seconds) is outlined in Column 6. The execution time was obtained on a 1.6GHz quad core Intel Core i5 with 4 GigaBytes of memory, running 32-bit Linux. The language used for the implementation was Java (OpenJDK 1.6.0_24).

L	k	m	$\lvert\Sigma\rvert$	Number of Equivalence Classes	Time (in sec)
10	1	4	2	4119	16.801
10	1	3	2	4637	17.681
9	1	3	2	2072	4.072
9	2	2	2	6303	46.479
8	4	3	2	3550	220.706
8	4	4	2	2346	49.867
8	3	2	2	4822	60.752
8	3	3	2	2736	33.95
8	2	2	2	2336	7.608
8	2	3	2	1501	5.056
7	1	2	2	918	1.512
7	1	3	2	395	0.484
6	1	2	2	248	0.3
6	1	2	3	2720	9.621
5	1	2	2	107	0.092
5	1	2	3	1007	1.576

TABLE I
Experimental Results

The results show that the proposed method works in reasonable time. It can be seen that the time for computing the strategy increases with L, k and the size of Σ. A higher value of L means that a longer history needs to be remembered, and hence, the state space will larger. k and $|\Sigma|$ affect the execution time by determining the branching factor of the Min-max tree. As k and $|\Sigma|$ increases, the branching factor increases, thereby increasing the execution time. On the other hand, the execution time decreases as the latency (m) increases. This is because more subtrees get pruned out, and thereby, less states need to be explored.

VII. Conclusion

We have described how latencies in fault detection may be handled in the automata-based scheduling framework. We have shown that the game theoretic approach can be used with certain modifications, in cases where faults are not immediately visible. Experimental results are found to be quite encouraging.

Acknowledgement

Pallab Dasgupta and P. P. Chakrabarti acknowledge the support of IGSTC (Indo-German Science and Technology Center) Project. Aritra Hazra is supported by Microsoft Corporation and Microsoft Research India under the Microsoft Research India PhD Fellowship Award.

References

[1] R. Alur and G. Weiss. Regular Specifications of Resource Requirements for Embedded Control Software. In *the Symposium on Real-Time and Embedded Technology and Applications (RTAS)*, pages 159–168, 2008.

[2] L. Gurvits. Stability of Discrete Linear Inclusion. *Linear Algebra Applications*, 231:47–85, 1995.

[3] J. P. Hespanha and A. S. Morse. Stability of Switched Systems with Average Dwell-time. In *Proceedings of the 38th IEEE Conference on Decision and Control*, volume 3, pages 2655–2660, 1999.

[4] D. Liberzon. *Switching in Systems and Control*. Birkhäuser, 2003.

[5] S. P. M, A. Hazra, and P. Dasgupta. Reliability Guarantees in Automata Based Scheduling for Embedded Control Software. *To appear in IEEE Embedded Systems Letters (An extended version of this is available at: http://facweb.iitkgp.ernet.in/∼pallab/TechRep/ESL2013TR.pdf)*, 2013.

[6] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Pearson Education, 2003.

[7] G. Weiss and R. Alur. Automata Based Interfaces for Control and Scheduling. In *10th International Workshop on Hybrid Systems: Computation and Control (HSCC)*, pages 601–613, 2007.