7-12-2005

Steering of Discrete Event Systems: Control Theory Approach

Arvind Easwaran
University of Pennsylvania, arvinde@seas.upenn.edu

Sampath Kannan
University of Pennsylvania, kannan@seas.upenn.edu

Oleg Sokolsky
University of Pennsylvania, sokolsky@cis.upenn.edu

Postprint version. Published in *Electronic Notes in Theoretical Computer Science*, Volume 144, Issue 4, 2005, pages 21-39.
Publisher URL: http://dx.doi.org/10.1016/j.entcs.2005.02.066

Steering of Discrete Event Systems: Control Theory Approach

Abstract

Runtime verification involves monitoring the system at runtime to check for conformance of the execution trace to user defined safety properties. Typically, run-time verifiers do not assume a system model and hence cannot predict violations until they occur. This limits the practical applicability of runtime verification. Steering is the process of predicting the occurrence of violations and preventing them by controlling system execution. Steerers can achieve this using a limited knowledge of the system model even in situations where it is infeasible to store the entire model. In this paper, we explore a control-theoretic view of steering for discrete event systems. We introduce an architecture for steering and also describe different steering paradigms.

Keywords

runtime correction, steering, runtime checking, control theory

Comments

Postprint version. Published in *Electronic Notes in Theoretical Computer Science*, Volume 144, Issue 4, 2005, pages 21-39.
Publisher URL: http://dx.doi.org/10.1016/j.entcs.2005.02.066

RV 2005 Preliminary Version

Steering of Discrete Event Systems: Control Theory Approach

Arvind Easwaran [1], Sampath Kannan [2] and Oleg Sokolsky [3]

Department of Computer and Information Science
University of Pennsylvania
Philadelphia, USA

Abstract

Runtime verification involves monitoring the system at runtime to check for conformance of the execution trace to user defined safety properties. Typically, run-time verifiers do not assume a system model and hence cannot predict violations until they occur. This limits the practical applicability of runtime verification. Steering is the process of predicting the occurrence of violations and preventing them by controlling system execution. Steerers can achieve this using a limited knowledge of the system model even in situations where it is infeasible to store the entire model. In this paper, we explore a control-theoretic view of steering for discrete event systems. We introduce an architecture for steering and also describe different steering paradigms.

Key words: runtime correction, steering, runtime checking, control theory

1 Introduction

Verification and validation are established techniques to ensure correctness of software systems. But, verification checks the specification rather than the implementation and validation does not provide guarantees for conformance. Also, verification and validation do not scale well to large systems. Runtime verification checks for conformance of execution traces to formally specified safety properties and hence exports the advantages of verification techniques to trace validation. Although runtime verifiers can efficiently detect non-conformance, they cannot predict violations in advance because they do not have information on the system model. Since any run-time technique deals

[1] Email: arvinde@seas.upenn.edu

[2] Email: kannan@cis.upenn.edu

[3] Email: sokolsky@cis.upenn.edu

This is a preliminary version. The final version will be published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

with a single execution trace, limited knowledge (local to the execution trace) of the system model will be sufficient to predict violations with high confidence. Steering exploits this limited system knowledge to predict violations.

A steerer looks ahead into a partial system model to explore the state space in advance. It then invokes the runtime verifier to detect non conformance and generates control actions to prevent the system from reaching a violation. Steering can thus be defined as the process of analysis (using look ahead) of a partial system model to detect non-conformance (prediction) and application of control actions to the model (prevention). We assume, the steerer and the system to be residing on different machines and executing concurrently. This leads to communication delay in the transmission of event notifications and steering actions between the steerer and the system.

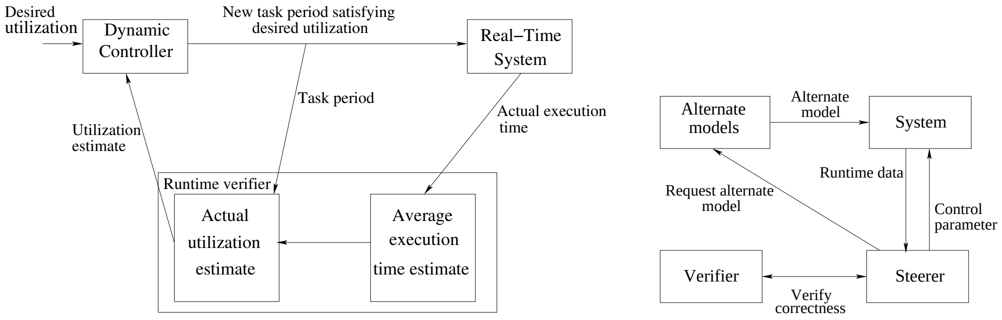
In this paper, we formulate the problem of steering of Discrete Event Systems (DES) as a supervisory control problem. Communication delay is accounted for by assuming partial observability of the system. We introduce a generic architecture for steering and propose four different steering paradigms. Constraints imposed on the system model and the steering overhead to the system are used to classify these paradigms. The paper then describes a subway system example to which a particular paradigm is applied.

Related Work: Stephane et. al. in their online control work [3], [4], [6] present a control-theory-based approach for control of DES with limited look ahead. We adapt this work to the steering problem where steerers must generate control actions with as little latency as possible in the presence of communication delays and concurrent execution. The steering framework described in [8], [10] is specific to a particular runtime verifier and assumes that a recovery after the occurrence of a violation is always possible. It assumes no knowledge of the system model and hence the user is required to specify steering actions. Papers [5], [7], [1] provide very domain specific solutions to simpler control problems and would be extremely inefficient in a general framework.

The rest of the paper is organized as follows. Section 2 lists motivating examples for steering and Section 3 formally defines the steering problem. Section 4 introduces the steering architecture and Section 5 formulates the steering problem as a control problem under partial observability. Section 6 describes four steering paradigms and Section 7 gives an illustrative example. Section 8 concludes the paper and indicates future research directions.

2 Motivation

When developing a software system, the designer and the users have at least an informal idea of properties that the system must satisfy. However, subtle errors throughout the software engineering process can lead to an incorrect implementation. Static analysis of an abstract model of the implementation to eliminate undesirable behaviors is a challenging problem when the state space



(a) Task scheduling

(b) Requirement adherence

Fig. 1. Steering for different domains

to be explored is large or time varying(dependent on external environment). Thus, in order to eliminate undesirable behaviors while preserving a rich set of allowable behaviors, we must use dynamic control. In this section, we list a few examples from different domains that support our claim for the need of a steerer.

2.1 Scheduling of real-time tasks using actual execution time

The paper [1] introduces a real-time task model in which the worst case execution times for each task, minimum and maximum execution periods for each task and the desired system utilization are given. The runtime verifier monitors task executions to determine their actual execution times and then evaluates the actual utilization of the system. The steerer uses this utilization estimate to adjust task periods so that the desired utilization may be achieved. This architecture is shown in Figure 1(a).

2.2 Reconciling System Requirements at run-time

At run-time, a software system can deviate from its expected behavior because of changes in the system environment. The paper [5] suggests two approaches for runtime steering. As shown in Figure 1(b), steering can be achieved by monitoring the system and modifying control parameters to minimize deviations. If the deviation is large, the steerer could also request that an alternate design model be used by the system. For example, consider a meeting scheduler scheduling a meeting based on the availability times of all the participants. One of the system requirements is that the scheduler should be able to get the availability times of the participants. One way of achieving this goal is by locating the agenda for each participant and extracting the availability times. An alternate approach would be to send email requests to participants. Steering would then involve switching between these approaches depending on the accessibility of the agenda. When using the email model, the steerer can also monitor the delay in response of participants and adjust reminder frequencies accordingly.

2.3 Inverted Pendulum

Paper [8] describes a control system which consists of an inverted pendulum run by an experimental controller. The experimental controller generates random values for the motor speed of the pendulum. The runtime monitor checks if the experimental controller is maintaining the pendulum within a stable region. If the monitor detects a violation it switches the control to a safety controller which maintains the pendulum in the stable region. This experiment demonstrates the use of steering in a continuous-variable, constrained environment.

3 Problem Statement

As described in Section 2 steering is essential to detect certain errors in software, especially when the system state space is large or time varying. In this paper, we develop a steering architecture and steering paradigms for DES with large state spaces.

3.1 Steering for DES with large state spaces

Any steerer for a DES must satisfy the following properties,

Light Weight: Steering must be light weight i.e., the steerer should not behave like a central controller.

Fast: Low response time is desirable because steering actions are state dependent. Low response time is also a desired property when steering real-time systems.

Minimally restrictive: Steering must restrict the system model minimally while ensuring that violations do not occur.

We now formulate the steering problem for DES with large state spaces. A DES is a transition system $M = \langle S, T, \Sigma \rangle$, where S is the large state space, Σ is the event set and T is the transition function $T : S \times \Sigma \longrightarrow S$. We assume that all the events Σ are controllable by the steerer. The state space S is discrete and hence finite or countably infinite. The state machine is event driven i.e., at each time instant at most one event can be generated. Let, n be the round-trip communication delay between the system and the steerer. This means that the system can generate a maximum of n events between the generation of an event e and the reception of steering action corresponding to e. We assume that the processing delay of the steerer is negligible compared to this delay. The steering problem can then be decomposed into the following problems: Given communication delay n, current state s'_0 of the DES M and the system model M,

(i) Predict non-conformance to user defined safety properties along different execution paths originating from s'_0 in the partial system model

(ii) Generate appropriate steering actions to ensure that the system does not reach violation by restricting the partial model

(iii) Determine a steering paradigm for execution of the steering actions by the system

Problem (i) can be formulated as a standard runtime verification problem once the partial system model has been generated. Problem (ii) has been addressed in this paper using supervisory control theory for DES. Steering paradigms designed later in the paper provide a solution to problem (iii). The same architecture and paradigms can be used for time varying systems if the partial system model for such systems can be generated at each step.

4 Steering Architecture for DES

The communication delay between the system and the steerer dictates the minimum amount of look ahead that the steerer must use to ensure that steering actions are transmitted to the system in time. Further, since steering actions are state dependent, each action must be executed by the system at the state for which that action was generated. In this section, we develop a steering architecture for DES which addresses these issues.

4.1 Steering Architecture

In response to an event notification, the steerer must generate a partial $(n+1)$ step DES originating from the current state as shown in Figure 2. $n+1$ is the minimum amount of look ahead required to ensure that steering actions are received by the system in time. The states of this DES are known as runtime states and each state uniquely identifies an execution trace in the system. Formally, the runtime DES for a current state s_0 and system model M can be defined as $G_{s_0} = \langle S', R, \Sigma, s_0, F, C, T' \rangle$. S' is the set of runtime states and R defines a function, mapping runtime states in S' to static states S in DES M. $s_0 \in S'$ is the initial state of G_{s_0} and Σ is the event set. $T' : S' \times \Sigma \longrightarrow S'$ is a transition function defined on G_{s_0} such that $T'(s', e) = T(R(s'), e)$ where $e \in \Sigma$. $F \subset S'$ is the set of final states of the runtime DES and $C \subset S'$ is the set of control states defined as $C = \{s' \in S' | \exists e \in \Sigma, f \in F, T'(s', e) = f\}$. The steering action for s_0 will disable some of the transitions with source states in C.

In response to a request from the steerer, the runtime verifier finds a set of violating states F' such that $F' \subseteq F$. Let, G'_{s_0} be the steered system for the original system G_{s_0} such that F' is not reachable in G'_{s_0}. Some steering paradigms might require that the steerer send activation and deactivation signals to the system for each steering action. These signals occur at points in the execution of the system and the steering action is applicable in the interval

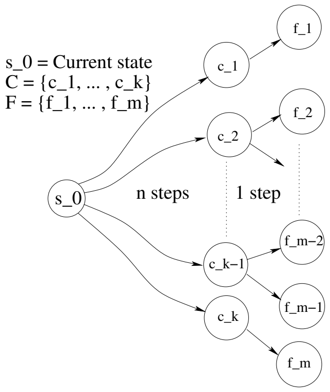


Fig. 2. Runtime DES G_{s_0}

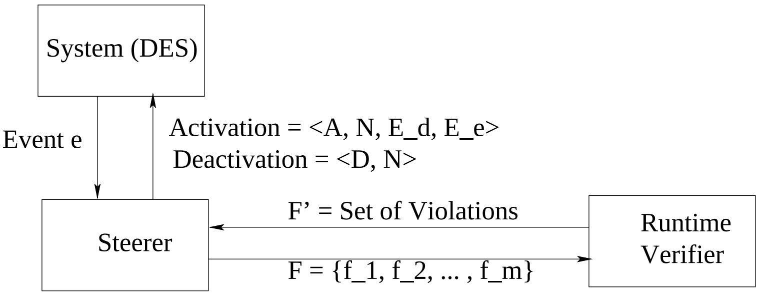


Fig. 3. Steering architecture

of time between these points. These signals are necessary because there is no synchronization between the system and the steerer and steering actions are associated with specific system states. An activation signal is of the form $\langle A, N, \Sigma_d, \Sigma_e \rangle$ where A implies activation and N is the action number. The steerer uses this number(unique for every activation-deactivation pair) when sending the deactivation signal. Σ_d is the set of events to be disabled and Σ_e is the set of events to be enabled in the system model. The deactivation signal is of the form $\langle D, N \rangle$ where D implies deactivation.

The architecture of a DES with a steerer and a runtime verifier is shown in Figure 3. The functions of each of the modules in the architecture is described below.

- **System(DES):** When an event is generated, the system sends an event notification to the steerer. On receipt of a steering action, the system may either apply the action immediately or store it for future use depending on the steering paradigm employed. The system may also receive activation and deactivation signals from the steerer. When the system receives an activation signal, it executes the accompanying steering action. When it receives a deactivation signal, it deactivates the action corresponding to the action number accompanying the signal.
- **Steerer:** On receiving an event notification from the system, the steerer generates the partial runtime DES G_{s_0}. It then sends the set of final states F

in G_{s_0} to the runtime verifier. For each state $u \in C$ having transitions to F', the steerer generates the set Σ_d^u of events that need to be disabled(transitions leading to F'). The steering action for the state s_0 is then given by $\Sigma_d = \bigcup_{u \in C} \Sigma_d^u$. Similarly, the steerer could also generate a set of enabled events Σ_e for C depending on the paradigm.

- **Monitor:** The monitor checks if any of the states in F(sent by the steerer) violate any of the user defined safety properties. It then reports all violations to the steerer in the set F'.

Steering problems defined in Section 3 can be refined under the architecture described in this section. Given communication delay n, current state s_0' and the system model M, steering entails

- finding the runtime DES $G_{s_0'}$,
- generating steering actions for $G_{s_0'}$. This would involve determining the violating states in $G_{s_0'}$ using the runtime verifier and
- determining a steering paradigm for the execution of steering actions.

Given system model M, current state s_0' and the steered runtime DES G_{s_0}' from the previous iteration, the steerer can efficiently compute the new runtime DES $G_{s_0'}$. If $e \in \Sigma$ is the current event received from the system, the steerer updates its runtime state to $s_0' = T'(s_0, e)$ where T' is the transition relation in G_{s_0}'. It then computes the states $f \in F \setminus F'$ of G_{s_0}' that are reachable from s_0'. $G_{s_0'}$ is now generated by determining the successor states of this reachable set using M.

5 Control Theory based Steering

The runtime DES G_{s_0} for which we wish to generate steering actions is a finite state machine. This steering problem can be formulated as a static control problem for DES under partial observation. The partial observation is as a result of the fact that the steerer has limited knowledge of the current state of the system due to communication delay.

5.1 Steering of runtime DES as a Static Control Problem

Let, G be the finite DES for which we wish to design a controller and $M = L(G)$ denote the language generated by G. $L(\bar{G}) = \{s \in \Sigma^* : (\exists t \in \Sigma^*) \wedge (st \in L(G))\}$ is the prefix closure of $L(G)$. Let, S denote the finite DES(supervisor) we wish to design such that $L(S \times G) \subseteq L(G)$ where $L(S \times G)$ is the language generated by the supervised system. $\Sigma_o \subseteq \Sigma$ denotes the set of observable events and $\Sigma_u = \Sigma \setminus \Sigma_o$ denotes the set of unobservable events. P is a projection function defined on event strings denoted as $P : \Sigma^* \longrightarrow \Sigma_o^*$. Figure 4 shows the architecture for control of finite DES under partial observation. The supervisor S can be defined by a function $SC : P(L(S \times G)) \longrightarrow 2^\Sigma$ given by $s \in L(S \times G) \wedge s\sigma \in L(G) \wedge \sigma \in SC(P(s)) \Rightarrow s\sigma \in L(S \times G)$.

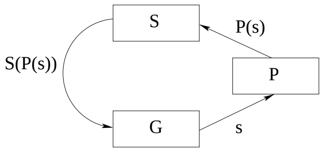


Fig. 4. Control under partial observation

Two different strings can be equivalent under partial observation and hence the control actions required for these two strings in order to generate the controlled language must be the same. The observability condition is used to determine whether a controlled language is admissible for a given system under given partial observability. Let, $P^i(s\sigma) = P^{-1}[P(s)]\sigma$ where $(s \in \Sigma^*) \wedge (\sigma \in \Sigma)$. Also, $P^i(\epsilon) = \epsilon$. Further, let $supP(N) = N \setminus [(\Sigma^* \setminus N)\Sigma^*]$ where $N \subseteq \Sigma^*$. Basically, $supP(N)$ is given by the set $\{s|(s \in N) \wedge (All\ prefixes\ of\ s\ are\ in\ N)\}$. Let K be any language such that $K \subseteq L(G)$ and $K \neq \phi$. We wish to decide if we can design a supervised system S generating $K(K = L(S \times G))$ and admissible for M under the projection P. The observability condition given in [2] can be stated as,

K is said to be observable with respect to M and P if
$\forall s \in \bar{K} \wedge \forall \sigma \in \Sigma, (s\sigma \notin \bar{K}) \wedge (s\sigma \in M) \Rightarrow P^{-1}[P(s)]\sigma \cap \bar{K} = \phi$

The observability condition given in [9] can be defined as,

K is observable with respect to M and P if and only if
$supP(P^i(\bar{K})) \cap M \subseteq \bar{K}$

Proof of equivalence of these two observability conditions is given in Appendix A. Steering paradigms described later in the paper will restrain the system model using these observability conditions to ensure correctness of the paradigm. Generation of steering actions for the runtime DES G_{s_0} can now be formulated as a static control problem under partial observation. G_{s_0} is the finite DES for which we wish to design a steerer and $G^{'}_{s_0}$ denotes the steered DES. The steering problem can now be stated as,

Given G_{s_0} and the set $F' \subseteq F$ of violating states, design $G^{'}_{s_0}$ such that F' is not reachable in the controlled system $G^{'}_{s_0}$. $G^{'}_{s_0}$ and G_{s_0} must satisfy observability conditions specified in the particular steering paradigm used.

6 Steering Paradigms

In this section we describe different steering paradigms that can be adopted under the architecture defined in Section 4. Since the steerer and the system have inconsistent views of the current state of the system, the steerer must assist the system in determining the state at which a particular steering action must be executed.

6.1 Paradigm 1: Unconstrained System Model(Hashing based Steering)

Steering paradigms under this classification do not impose any restrictions on the system model. Hashing based execution of steering actions is one such paradigm. In this approach, the steerer sends hashed values of event strings(states u in C that have transitions to violating states in F) and the corresponding set of disabled events(Σ_d^u) to the system. The system stores these values in a hash table. When an event occurs, the system generates a new hash value(incrementally) for the event string seen so far and hashes into this table. On a hit, the system activates the steering action associated with that slot. This action can be deactivated on the occurrence of any further event.

We now design a hash function for this paradigm. Let, $(k-1)$ denote the number of different events in the system model. We will then use a radix-k representation for event strings. Each event $e \in \Sigma$ will be associated with a unique integer $i_e \in (1, k-1)$. For every event string $es = e_1 e_2 \cdots e_l$ where $i_{e_1}, i_{e_2}, \cdots, i_{e_l}$ are the unique integers for events $e_1, e_2, \cdots, e_l$ respectively, its radix-k representation $r_{es} = i_{e_l} + i_{e_{l-1}} \times k + \cdots + i_{e_1} \times k^{l-1}$. Given es and its radix-k representation r_{es}, the radix-k representation of the event string $es' = es.e_{l+1}$(. represents string append) is $r_{es'} = i_{e_{l+1}} + (r_{es} \times k)$.

This representation splits event strings into different ranges of natural numbers depending on string lengths. The hash function we design, will ensure that event strings with different lengths do not hash into the same slot. We split the entire hash table into n regions numbered $0, \cdots, (n-1)$ each of the same size. Event string $es \in \Sigma^*$ will hash into the $(|es| \ mod \ n)^{th}$ region in the hash table where $|es|$ denotes the string length. Splitting the hash table into n regions is sufficient because n is the communication delay. When steering action for a string of length $2n$ is received by the system, the system must have executed the action for strings of length n(steerer generates steering actions for strings of length $2n$ after the system reaches states corresponding to strings of length n). All strings of length less than n cannot be associated with any steering action and hence will hash into a single slot.

Let, $m+1$ be the total size of the hash table where m/n is an integer. The size of each region is m/n and slot 0 will be used to hash all strings of length less than n. The hash function is then given by,

$$\begin{aligned} h(r_{es}, i) \quad &= 0 && i \leq (n-1) \\ &= (r_{es} \bmod m/n) + ((i \bmod n)(m/n)) + 1 && \text{otherwise} \end{aligned}$$

The relation between r_{es} and i in the hash function is given by $(k^i - 1)/(k-1) \leq r_{es} \leq (k^i - 1)$. The average chain length at each slot is $\alpha = O(((k-1)^n)n/m)$ because the maximum number of strings that can hash into a single region is always $(k-1)^n$(steerer generates steering actions for runtime states in C using G_{s_0} and has complete knowledge of the execution trace upto s_0. The length of any event string originating from s_0 and terminating in some state in C is always n).

Splitting the hash table into n regions up front can lead to under utilization of the hash table. The modified hash function which splits the hash table dynamically is given by,

$$
\begin{array}{lll}
h(r_{es}, i, j) & = 0 & i \leq (n-1) \\
 & = (r_{es} \bmod m/j) + ((i \bmod n)(m/j)) + 1 & \text{otherwise}
\end{array}
$$

The steerer will now send the entire hash table to the system every time it generates new steering actions. In practice, violating states are a small subset of the state space and hence the number of entries in the hash table will be small. The steerer also sends the number of regions j into which the hash table is currently split.

Advantages This paradigm does not impose any restrictions on the system model.

Disadvantages There is an overhead on the system for storing the hash table and for computing the hash function.

6.1.1 System correctness

System correctness depends on the interaction between event strings that participate in a collision in the hash table. A conflict occurs between two colliding strings only when an event disabled by the steering action of one string is required to be enabled in the other. We make a pessimistic assumption that two colliding strings lead to incorrect control when either of them have any steering action associated with them. Let, h denote the number of hash slots in each region of the hash table, p the average number of strings hashing into a region and d the average number of strings that have steering actions associated with them. Let the set of event strings that have steering actions associated be denoted by D. Assuming simple uniform hashing, the probability of collision between any two strings r_{es1}, r_{es2} hashing into the same region and having same length i is $Prob\{h(r_{es1}, i) = h(r_{es2}, i)\} = 1/h$. The probability that the system during its execution generates one of the strings from r_{es1} and r_{es2} is $Prob\{$ System generates $r_{es1} \vee r_{es2}\} = 2/p$. Using our assumption, the probability that r_{es1} and r_{es2} conflict is given by,

$$Prob\{r_{es1}, r_{es2} \text{ conflict } |(r_{es1} \in D) \vee (r_{es2} \in D)\} = 1$$
$$Prob\{r_{es1}, r_{es2} \text{ conflict } |(r_{es1} \notin D) \wedge (r_{es2} \notin D)\} = 0$$

Given n is the number of regions in the hash table and $Prob\{Region\ fails\}$ is the probability of failure(colliding and conflicting strings) in one region, the probability that the steered system is correct is,

$$Prob\{System\ correct\} = [1 - Prob\{Region\ fails\}]^n$$

Now, $Prob\{Region\ fails\} = Prob\{h(r_{es1}, i) = h(r_{es2}, i)\} \times Prob\{$ System generates $r_{es1} \vee r_{es2}\} \times Prob\{r_{es1}, r_{es2}$ conflict $\}$

$= 1/h \times 2/p \times [Prob\{r_{es1}, r_{es2}$ conflict $|r_{es1} \in D$ or $r_{es2} \in D\}\times$ $Prob\{r_{es1} \in D$ or $r_{es2} \in D\} + Prob\{r_{es1}, r_{es2}$ conflict $|r_{es1} \notin D$ and $r_{es2} \notin D\}\times$ $Prob\{r_{es1} \notin D$ and $r_{es2} \notin D\}]$

$= 1/h \times 2/p \times [Prob\{r_{es1} \in D$ or $r_{es2} \in D\} + 0]$
$= 1/h \times 2/p \times [1 - Prob\{r_{es1} \notin D$ and $r_{es2} \notin D\}]$
$= 2/(hp)[1- \binom{p-d}{2} \ / \ \binom{p}{2}]$

Therefore, $Prob\{Region\ fails\} = 2/(hp)[d(2p-d-1)/p(p-1)]$
Hence, $Prob\{System\ correct\} = [1-(2/(hp)[d(2p-d-1)/p(p-1)])]^n$

6.2 Constrained System Model

In this steering paradigm we reduce the steering overhead on the system by constraining the system model. A constrained model will provide opportunities for automated activation and deactivation of steering actions. Let, $P_{n_1,n_2}(s) = s.s'$ such that $n_1 \leq |s'| \leq n_2$ where $s, s' \in \Sigma^*$ and $|s'|$ denotes length of string s'. Also, let G_{s_0} be the original system, F be the final states of G_{s_0} and $G^{'}_{s_0}$ be the steered system.

6.2.1 Paradigm 2: Event Trace based Steering

In this approach, the steerer sends the disabled event set Σ_d and its observed event trace s_0 to the system. The system compares its own event trace with the trace sent by the steerer and determines the distance to C(states where steering actions must be executed) for that steering action. It stores this distance along with the steering action and decrements the distance at the occurance of each event. When this distance reduces to zero, the corresponding steering action is executed. Deactivation can be done at the occurance of any further event. When the steering action is executed, the system could be in any state in C. To ensure correctness, the system model must satisfy the observability condition given in Section 5.1 where $\bar{K} = G^{'}_{s_0}$, $M = G_{s_0}$ and all strings in G_{s_0} are equivalent under projection P.

Advantages This paradigm imposes less overhead on the system as compared to hashing.

Disadvantages The system must store its event trace and also update the distance metric for all the stored steering actions at the occurance of every event. The system model is also constrained by the observability condition.

6.2.2 Paradigm 3: Steering with Observability Condition and Minimum Separation

In this paradigm, the system will disable all the events in Σ_d as soon as it receives the steering action $\langle A, N, E_d, E_e\rangle$($N$ not required). To ensure admissibility of the steering action, the system model must satisfy the observability condition as in paradigm 2. The system will deactivate the steering action as soon as any event in $\Sigma_d \cup \Sigma_e$ occurs. A minimum separation property of events in the model will ensure correctness of this deactivation procedure. Minimum separation is given by,

$$\forall s\sigma \in (G_{s_0} \cap F), P_{0,n-1}[s_0]\sigma \cap G_{s_0} = \phi$$

This property ensures that no event in $\Sigma_d \cup \Sigma_e$ will occur in any of the transitions with source states in $S' \setminus (F \cup C)$.

Advantages Event traces are not required to be stored by the system in this paradigm. The steering overhead on the system is also minimal.

Disadvantages The steerer is required to send both the event sets Σ_d and Σ_e. The system model is also severely constrained.

6.2.3 Paradigm 4: Steering with Stringent Observability Condition

In this approach, the system will disable all the events in Σ_d as soon as it receives the steering action $\langle A, N, \Sigma_d, \Sigma_e \rangle(\Sigma_e$ is empty). When the steerer sends the activation signal, it stores the action number N alongwith the corresponding set of final states F of $G^{'}_{s_0}$. When the steerer receives an event notification, it checks if the current state matches with any state in F for any action number. If such a match occurs for action number N, the steerer sends the deactivation signal $\langle D, N \rangle$. On receiving a deactivation signal, the system deactivates the steering action specified by the action number. To ensure admissibility of steering actions, the system model must satisfy a modified observability condition given by,

$$\forall s \in G^{'}_{s_0}, \forall \sigma \in \Sigma, \ (s\sigma \notin G^{'}_{s_0}) \wedge (s\sigma \in G_{s_0}) \Rightarrow P_{0,2n}[s_0]\sigma \cap G^2_{s_0} = \phi \text{ where}$$

$G^2_{s_0}$ is the steered runtime DES originating from s_0 and having depth $2n + 1$. When the system receives the activation signal, it could be in any of the states in $S' \setminus F$ and when it receives the deactivation signal it could be in any state which is at a distance of atmost n steps from some state in F. The modified observability condition ensures that if $\sigma \in \Sigma_d$, then all the σ transitions in this entire region are disabled.

Advantages This paradigm induces minimal steering overhead on the system. The restriction on the system model is less stringent than in paradigm 3.

Disadvantages The steerer has to send deactivation signals and also needs additional storage for previously sent activation signals.

7 Illustrative Example

In this section, we apply steering paradigm 4 to an example of a simple subway system [3]. We demonstrate the ability of our architecture to prevent the subway system from entering violations and highlight the constraints that the

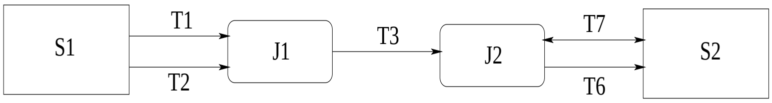


Fig. 5. Subway System

steering paradigm enforces on the system model.

7.1 Simple Subway System

As shown in Figure 5, a subway system consists of stations, junctions, trains and a set of tracks with directions. Each train is represented by its source station, destination station and a route. We assume each track is further divided into 4 sections. This is a good example for dynamic control because the state space grows with the number of trains in the system and this in general can be very large.

Safety properties for this subway system may include,

(i) Limit on the maximum number of trains at any given time at any junction. For example, let one safety property be "Junction J1 has capacity for only 2 trains at any given time"

(ii) Balanced usage of tracks in the system. For example, let the system be constrained by the property that "The difference in loads(number of trains that have used the track) in tracks T6 and T7 must never exceed 4".

We would like to steer the system and prevent it from violating any of these safety properties. We assume that the round-trip communication delay between the steerer and the subway system is $n = 1$.

7.2 Steering of Subway System

Let, $\alpha(i, j, k)$ represent the event that train k enters section j of track i and $\alpha(i, 5, k)$ represent the event that train k left track i and entered a junction or a station. Let for example, the system consist of three trains X, Y and Z. Assume train X executed event $\alpha(1, 4, X)$ and Y executed event $\alpha(2, 4, Y)$ most recently. Let train Z be currently at junction $J1$ and let s represent this state of the combined system. At this state, the system can violate safety property 1 in $n + 1 = 2$ steps(both X and Y decide to enter $J1$ without train Z leaving it). Hence the steerer, under paradigm 4, will disable events $\alpha(1, 5, X)$ and $\alpha(2, 5, Y)$ at state s as shown in Figure 6. Train Z could move to section 1 of track $T3$ before trains X and Y arrive at junction $J1$ and this will ensure that property 1 is not violated. But for admissibility of the steering action under paradigm 4, the system must satisfy observability condition specified in section 6.2.3. This condition would require that events $\alpha(1, 5, X)$ and $\alpha(2, 5, Y)$ be disabled in all states which are at most $n + 1 = 2$ steps from s as shown in Figure 6. If we use hashing based steering(Paradigm

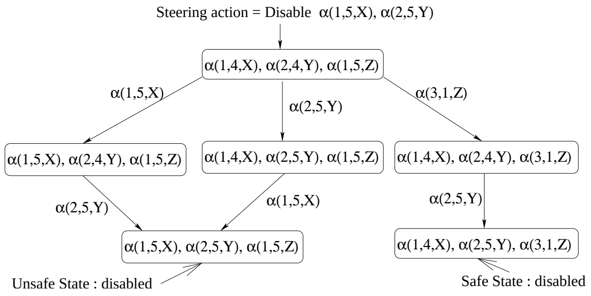

Steering action = Disable α(1,5,X), α(2,5,Y)
α(1,4,X), α(2,4,Y), α(1,5,Z)
α(1,5,X)
α(2,5,Y)
α(3,1,Z)
α(1,5,X), α(2,4,Y), α(1,5,Z)
α(1,4,X), α(2,5,Y), α(1,5,Z)
α(1,4,X), α(2,4,Y), α(3,1,Z)
α(2,5,Y)
α(1,5,X)
α(2,5,Y)
α(1,5,X), α(2,5,Y), α(1,5,Z)
α(1,4,X), α(2,5,Y), α(3,1,Z)
Unsafe State : disabled
Safe State : disabled

Fig. 6. Paradigm 4 applied to Safety Property 1

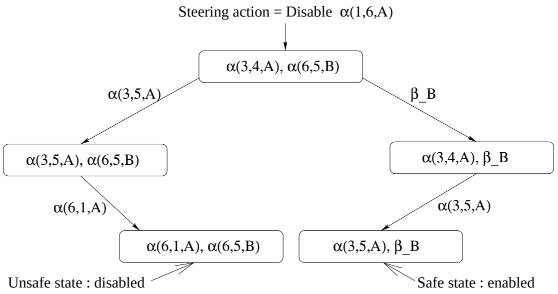


Fig. 7. Paradigm 4 applied to Safety Property 2

1), valid executions are no longer required to be disabled but this technique does induce additional overhead on the system.

Let, β_k represent the event that train k enters track $T7$ from $S2$ and γ_k be the event that train k enters track $T7$ from $J2$. Safety property 2 can now be written as $-4 \leq (|\beta| + |\gamma| - |\alpha(6,1,*)|) \leq 4$ where $|e|$ represents the number of occurrences of event e(ignoring train numbers) in the system execution history. Let, the system currently consist of trains A and B only where train A has executed event $\alpha(3,4,A)$ most recently and train B is at station $S2$. Also, let the current value of $(|\beta| + |\gamma| - |\alpha(6,1,*)|)$ be -4. Now, if train A enters track $T6$ before train B enters track $T7$, safety property 2 will be violated. To prevent this, the steerer will disable event $\alpha(6,1,A)$ at the current state. The observability condition in section 6.2.3 does not block safe states in this case as shown in Figure 7.

8 Conclusion and Future Work

In this paper, we have introduced an architecture for steering of DES with large state spaces and also described different steering paradigms for execution of steering actions. Runtime monitors equipped with steerers will be able to predict occurrence of violations well in advance to be able to steer the system away from them. The paper, essentially views the steerer as a dynamic controller that uses the runtime verifier as a sub-module to predict violations. If the violations are restricted to a small subset of the state space, then this architecture will provide efficient steering for DES with large state spaces.

Generation of the partial system model for time varying systems is an important extension for this steering architecture. Extending the steering architecture to models where not all transitions are controllable is also a natural area to explore because in a typical system not all execution points are controllable. Reducing steering overhead by using probabilistic prediction of violations is another direction for further work.

References

[1] Giorgio Buttazzo and Luca Abeni. Adaptive workload management through elastic scheduling. *Real-Time Systems*, 23(1-2):7–24, 2002.

[2] Christos G. Cassandras and Stephane Lafortune. *Introduction to Discrete Event Systems*. Kluwer Academic Publishers, Norwell, Massachussetts, 1999.

[3] Sheng-Luen Chung, Stephane Lafortune, and Feng Lin. Limited lookahead policies in supervisory control of discrete event systems. In *IEEE Transactions on Automatic Control*, volume 37. IEEE, 1992.

[4] Sheng-Luen Chung, Stephane Lafortune, and Feng Lin. Supervisory control using variable lookahead policies. In *Proceedings of Discrete Event Dynamic Systems*. Kluwer, 1994.

[5] M. S. Feather, S. Fickas, A. van Lamsweerde, and C. Ponsard. Reconciling system requirements and runtime behaviour. *9th International Workshop on Software Specification and Design*, Apr 1998.

[6] Nejib Ben Hadj-Alouane, Stephane Lafortune, and Feng Lin. Variable lookahead supervisory control with state information. In *IEEE Transactions on Automatic Control*, volume 39. IEEE, 1994.

[7] Yinghua Jia and Joanne M. Atlee. Run-time management of feature interactions. *ICSE Workshop on Component based Software Engineering*, May 2003.

[8] Moonjoo Kim, Insup Lee, Usa Sammapun, Jangwoo Shin, and Oleg Sokolsky. Monitoring, checking, and steering of real-time systems. In *Electronic Notes in Theoretical Computer Science*, volume 70. Elsevier, 2002.

[9] Ratnesh Kumar and Vijay K. Garg. *Modeling and Control of Logical Discrete Event Systems*. Kluwer Academic Publishers, Norwell, Massachussetts, 1995.

[10] Oleg Sokolsky, Sampath Kannan, Moonjoo Kim, Insup Lee, and Mahesh Viswanathan. Steering of real-time systems based on monitoring and checking. *Fifth International Workshop on Object-Oriented Real-Time Dependable Systems*, 1999.

A Equivalence of Observability Conditions

The observability condition given in [2] can be stated as,
K is said to be observable with respect to *M* and *P* if

$$\forall s \in \bar{K} \wedge \forall \sigma \in \Sigma, (s\sigma \notin \bar{K}) \wedge (s\sigma \in M) \Rightarrow P^{-1}[P(s)]\sigma \cap \bar{K} = \phi \tag{A.1}$$

The observability condition given in [9] can be defined as,
K is observable with respect to *M* and *P* if and only if

$$supP(P^i(\bar{K})) \cap M \subseteq \bar{K} \tag{A.2}$$

We first prove some lemmas which will then be used to prove the equivalence of Eq. (A.1) and Eq. (A.2).

Lemma A.1 *Eq.* (A.1) $\Leftrightarrow [\forall s\sigma, \sigma \in \Sigma, [(s \in \bar{K}) \wedge (s\sigma \in M) \wedge (\exists s'\sigma \in \bar{K}, s'\sigma \in P^i(s\sigma))] \Rightarrow (s\sigma \in \bar{K})]$

Proof. From Eq. (A.1) we have,
$\forall s \in \bar{K}$ and $\forall \sigma \in \Sigma$, $(s\sigma \notin \bar{K}) \wedge (s\sigma \in M) \Rightarrow P^{-1}[P(s)]\sigma \cap \bar{K} = \phi$
$\Leftrightarrow \forall s \in \bar{K}$ and $\forall \sigma \in \Sigma$, $(s\sigma \notin \bar{K}) \Rightarrow (s\sigma \notin M) \vee (\forall s'\sigma \in \bar{K}, s'\sigma \notin P^{-1}[P(s)]\sigma)$
$\Leftrightarrow \forall s \in \bar{K}$ and $\forall \sigma \in \Sigma$, $((s\sigma \in M) \wedge (\exists s'\sigma \in \bar{K}, s'\sigma \in P^i(s\sigma)) \Rightarrow (s\sigma \in \bar{K}))$

$$\Leftrightarrow \forall s\sigma, \sigma \in \Sigma, ((s \in \bar{K}) \wedge (s\sigma \in M) \wedge (\exists s'\sigma \in \bar{K}, s'\sigma \in P^i(s\sigma)) \Rightarrow (s\sigma \in \bar{K})) \tag{A.3}$$

Thus Eq. (A.1) $\Leftrightarrow$ Eq. (A.3) and this proves Lemma A.1 □

Lemma A.2 $s\sigma \in P^i(s'\sigma) \Leftrightarrow s'\sigma \in P^i(s\sigma)$

Proof. We now prove that $s\sigma \in P^i(s'\sigma)$ implies $s'\sigma \in P^i(s\sigma)$. From the definition of $P^i(.)$,

$s\sigma \in P^i(s'\sigma) \Rightarrow s \in P^{-1}[P(s')] \Rightarrow s' \in P^{-1}[P(s)] \Rightarrow s'\sigma \in P^{-1}[P(s)]\sigma \Rightarrow s'\sigma \in P^i(s\sigma)$.

The other direction can be proved similarly. □

Lemma A.3 *Eq.* (A.3) $\Leftrightarrow \forall s\sigma, \sigma \in \Sigma$, $((s \in \bar{K}) \wedge (s\sigma \in M) \wedge (s\sigma \in supP(P^i(\bar{K}))) \Rightarrow (s\sigma \in \bar{K}))$

Proof. From Eq. (A.3) we have,
$\forall s\sigma, \sigma \in \Sigma$, $((s \in \bar{K}) \wedge (s\sigma \in M) \wedge (\exists s'\sigma \in \bar{K}, s'\sigma \in P^i(s\sigma)) \Rightarrow (s\sigma \in \bar{K}))$

From Lemma A.2 we have,
$\Leftrightarrow \forall s\sigma, \sigma \in \Sigma$, $((s \in \bar{K}) \wedge (s\sigma \in M) \wedge (\exists s'\sigma \in \bar{K}, s\sigma \in P^i(s'\sigma)) \Rightarrow (s\sigma \in \bar{K}))$
$\Leftrightarrow \forall s\sigma, \sigma \in \Sigma$, $((s \in \bar{K}) \wedge (s\sigma \in M) \wedge (s\sigma \in P^i(\bar{K})) \Rightarrow (s\sigma \in \bar{K}))$

$\bar{K}$ is prefix closed and hence we get,

$$\Leftrightarrow \forall s\sigma, \sigma \in \Sigma, ((s \in \bar{K}) \wedge (s\sigma \in M) \wedge (s\sigma \in supP(P^i(\bar{K}))) \Rightarrow (s\sigma \in \bar{K})) \tag{A.4}$$

Thus Eq. (A.3) $\Leftrightarrow$ Eq. (A.4) and this proves Lemma A.3. □

We now prove the following theorem,

Theorem A.4 *The observability conditions given by Eq.* (A.2) *and Eq.* (A.1) *are equivalent.*

Proof. Equation (A.2) $\Rightarrow$ **Equation** (A.1)

From Eq. (A.2) we have, $supP(P^i(\bar{K})) \cap M \subseteq \bar{K}$
$\Rightarrow (\forall s\sigma, \sigma \in \Sigma,\ [(s\sigma \in (supP(P^i(\bar{K})))) \wedge (s\sigma \in M)] \Rightarrow (s\sigma \in \bar{K}))$

Now, $s\sigma \in (supP(P^i(\bar{K}))) \Rightarrow \exists s'\sigma \in \bar{K}, s\sigma \in P^i(s'\sigma) \Rightarrow \exists s'\sigma \in \bar{K}, s'\sigma \in P^i(s\sigma)$

Also, $\forall t$ such that t is a prefix of $s\sigma$, $(t \in M) \wedge (t \in supP(P^i(\bar{K})))$. Using this and Eq. (A.2) we get, $s \in \bar{K}$.

Therefore Eq. (A.2) implies that,
$\forall s\sigma, \sigma \in \Sigma,\ ((s \in \bar{K}) \wedge (s\sigma \in M) \wedge (\exists s'\sigma \in \bar{K}, s'\sigma \in P^i(s\sigma)) \Rightarrow (s\sigma \in \bar{K}))$

But this is Eq. (A.3) and using Lemma A.1 we have Eq. (A.2) $\Rightarrow$ Eq. (A.1).

Equation (A.1) $\Rightarrow$ **Equation** (A.2)

From Lemma A.1 and A.3 we have, Eq. (A.1) is equivalent to Eq. (A.4). Hence to prove Eq. (A.1) $\Rightarrow$ Eq. (A.2) we prove Eq. (A.4) $\Rightarrow$ Eq. (A.2). i.e., we prove $\neg$Eq. (A.2) $\Rightarrow$ $\neg$Eq. (A.4).

From Eq. (A.2) we have, $supP(P^i(\bar{K})) \cap M \subseteq \bar{K}$
$\Leftrightarrow [\forall s\ [(s \in (supP(P^i(\bar{K})))) \wedge (s \in M)] \Rightarrow (s \in \bar{K})]$

Therefore, $\neg$Eq. (A.2) is given by,
$\exists s, (s \in (supP(P^i(\bar{K})))) \wedge (s \in M) \wedge (s \notin \bar{K})$
$\Rightarrow \exists s'\alpha, \alpha \in \Sigma, (s'\alpha \notin \bar{K}) \wedge (s' \in \bar{K}) \wedge (s'\alpha \in M) \wedge (s'\alpha \in supP(P^i(\bar{K})))$
where

$s'\alpha$ is the smallest prefix of s such that $s' \in \bar{K}$ and $s'\alpha \notin \bar{K}$(since $s \notin \bar{K}$, $s \in M$ and $s \in supP(P^i(\bar{K}))$ such a prefix exists).

This is equivalent to $\neg$Eq. (A.4). Hence, $\neg$Eq. (A.2) $\Rightarrow$ $\neg$Eq. (A.4) which using Lemma A.1 and A.3 gives Eq. (A.1) $\Rightarrow$ Eq. (A.2). □