# Supervisory Control of Software Systems

Vir V. Phoha, *Senior Member*, *IEEE*, Amit U. Nadgar,
Asok Ray, *Fellow*, *IEEE*, and Shashi Phoha, *Senior Member*, *IEEE*

**Abstract**—We present a new paradigm to control software systems based on the Supervisory Control Theory (SCT). Our method uses the SCT to model the execution of a software application by restricting the actions of the OS with little or no modifications in the underlying OS. Our approach can be generalized to any software application as the interactions of the application with the OS are modeled at a *process* level as a Deterministic Finite State Automaton (DFSA) termed as a "plant." A "supervisor" that controls the plant is a DFSA synthesized from a set of control specifications. The supervisor operates synchronously with the plant to restrict the language accepted by the plant to satisfy the control specifications. Using the above method of *control* to mitigate faults, as a proof-of-concept, we implement two supervisors under the Redhat Linux 7.2 OS to mitigate overflow and segmentation faults in five different programs. We quantify the performance of the unsupervised and supervised plant by using a Language Measure and give methods to compute the measure using *state transition cost* matrix and *characteristic vector*.

**Index Terms**—Systems and software, control theory, fault tolerance, automata, languages.

## 1 Introduction

A computer program is a discrete-event system in which the supervisory control theory (SCT) [1] can be applied to augment a general-purpose operating system (OS) to control and direct a wide range of software applications. We propose a novel SCT-based technique, built upon formal language theory, to model and control software systems without any structural modifications in the underlying OS. In this setting, the user has the privilege to override the OS actions to control a software application.

SCT is a well-studied paradigm and has been used in a variety of applications. However, for the sake of completeness, we very briefly review some relevant applications of SCT to software systems. Self-adaptation in software systems where supervisory control is augmented with an adaptive component is reported in [2]. SCT has been used in the Workflow management paradigm to schedule concurrent tasks through scheduling controllers [3] and also for protocol converters to ensure consistent communications in heterogeneous network environments. Hong et al. [4] has adopted supervisor-based closed-loop control to facilitate software rejuvenation—a technique to improve software reliability during its operational phase.

Ramadge and Wonham [1] present a novel SCT approach to control discrete event systems (DES) using a feedback control mechanism. Here, the DES to be controlled is modeled by the plant automaton $\mathcal{G}$. The uncontrolled behavior of the plant is modified by a supervisor $S$ such that behavior of the plant is restricted to a subset of $L(\mathcal{G})$. The feedback control that is achieved using the supervisor satisfies a given set of specifications interpreted as sublanguages of $L(\mathcal{G})$ representing legal behavior for the controlled system. Following the approach, this paper models interactions of software applications with the OS as a deterministic finite state automaton (DFSA) (i.e., a representation for the class of regular language) [6] and applies the SCT for development of a recognizer of this language to control and mitigate faults in software execution. Specifically, the discrete-event supervisor restricts the legal language of the model in an attempt to mitigate the normally detrimental consequences of faults or undesirable events.

In our approach, we first enumerate the DFSA's states and the events of the plant model $\mathcal{G}$. The specifications to control (restrict) the behavior of a computer program by controlling the interactions with the OS are represented by another DFSA $S$ that has the same event alphabet as the plant model. The parallel combination of $S$ and $\mathcal{G}$ gives rise to a DFSA $(S/\mathcal{G})$, which is the generator under supervisory control [1].

The significant contributions of the decision and control approach proposed in this paper are:

1. a novel technique for fault mitigation in software systems,
2. real-time control of software systems,
3. runtime behavioral modification and control of the OS with insignificant changes in the underlying OS,
4. modeling of the OS-Application interactions as symbols (events) in the formal language setting,
5. accommodation of multiple control policies by varying the state transitions,
6. from the perspectives of fundamental research, this work introduces the concept of discrete-event supervisory control theory to software systems. From the application perspectives, it provides new approach to software fault mitigation.

A few other researchers (see, for example, [7], [8], [9]) report monitoring of programs at runtime using automata and, in one case, *transforming the sequence when it deviates from the specified policy* [7]. However, the approach in [7], [8], [9] is significantly different from that taken in our work, where we have used novel principles of supervisory control theory as an extension of Ramadge and Wonham's work [1]. This concept results in parallel, synchronized operation of two or more automata, namely, a plant (i.e., the application or the computer program) and a controller (or a set of controllers), which implements the supervisory control policy on the entir plant rathern than on selected components.

- *V.V. Phoha and A.U. Nadgar are with Louisiana Tech University, Ruston, LA 71272. E-mail: phoha@latech.edu, rangekutta@hotmail.com.*
- *A. Ray and S. Phoha are with Pennsylvania State University, University Park, PA 16802. E-mail: {axr2, sxp26}@psu.edu.*

*Manuscript received 7 Sept. 2003; revised 29 Mar. 2004; accepted 6 Apr. 2004.*
*For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number TC-0147-0903.*

The rest of the paper is organized as follows: We review the Supervisory Control Theory in Section 2. In Section 3, we present discrete-event modeling of interactions of a computer process and the OS. Modeling of two supervisors based on a given specification is shown in Section 4. Section 5 describes an implementation of the supervisory control system for process execution under Red Hat Linux 7.2. In Section 6, we review the Language Measure Theory [10], [11], [12] and give a procedure for estimation of the event cost $\tilde{\Pi}$-matrix parameters in Section 7. A description of the experiments and the corresponding results are given in Section 8. We summarize and conclude our work in Section 9. In Appendix A, we give the definitions pertinent to Supervisory Control Theory. In Appendix B, we present a theoretical bound on the number of experimental observations necessary for identification of the language measure parameters.

## 2 Background

In this section, we review the supervisory control theory (SCT) of Discrete Event Systems [1], [13], [14]. A discrete event system (DES) is a dynamical system which evolves due to asynchronous occurrences of certain discrete changes called *event*. A DES has discrete states which correspond to some continua in the evolution of a task. The state transitions in such systems are produced at asynchronous discrete instants of time in response to events and represent discrete changes in the task evolution.

The SCT introduced by Ramadge and Wonham [1] is based on automata and formal language models. Under these models the focus is on the order in which the events occur. A plant is assumed to be the generator of these events. The behavior of the plant model describes event trajectories over the (finite) event alphabet $\Sigma$. These event trajectories can be thought of as strings over $\Sigma$. Then, $L \subseteq \Sigma^*$ represents the set of those event strings that describe the behavior of the plant. In this formal language setting, the concepts of plant and supervisor are discussed in the following subsections.

### 2.1 Plant Model

The plant $\mathcal{G}$ is a generator of all the strings in $L$ and is described as a quintuple deterministic finite state automaton (DFSA)

$$\mathcal{G} = (Q, \Sigma, \delta, q_0, Q_m), \tag{1}$$

where $Q$ is the set of states for the system with $|Q| = n$, $q_0$ is the initial state, $\Sigma$ is the (finite) alphabet of events causing the state transitions, $\Sigma^*$ is the set of all finite-length strings of events including the empty string $\epsilon$. The state transition function is defined as: $\delta : Q \times \Sigma \to Q$ and $\delta^* : Q \times \Sigma^* \to Q$ is an extension of $\delta$ which can be defined recursively as follows:

For any $q \in Q$, $\delta^*(q\epsilon) = q$; for any $s \in \Sigma^*$, $a \in \Sigma$, and $q \in Q$, $\delta^*(q, sa) = \delta(\delta^*(q, s), a)$; $Q_m \subseteq Q$ is the subset of states called the marked (or accepted) states.

A marked state represents the completion of a task or a set of tasks by the physical system we model. To give this system a means of control, the event alphabet $\Sigma$ is classified into two categories: **uncontrollable events** ($\sigma \in \Sigma_{uc}$), which can be observed but cannot be prevented from occurring, and **controllable events** ($\sigma \in \Sigma_c$), which can be prevented from occurring.

### 2.2 Supervisor

A supervisor $S$ is realized as a function $S = (\mathcal{S}, \phi)$. $\mathcal{S}$ is given by the DFSA quintuple:

$$\mathcal{S} = (X, \Sigma, \xi, x_0, X_m), \tag{2}$$

where $X$ is the state set, $\Sigma$ is the event alphabet which is same as that of the plant, $\xi : \Sigma \times X \to X$ is the transition function, and $X_m \subseteq X$ is a subset of marked states, $\phi$ is a function that maps the states of the supervisor into control patterns $\gamma \in \Gamma$ where $\Gamma = \{0,1\}^{\Sigma_c}$ is the set of all binary assignments to the elements of $\Sigma_c$. Each state of the supervisor corresponds to a fixed control pattern where some controllable events are enabled or disabled. Thus, the plant is controlled by the supervisor by switching to control patterns corresponding to the supervisor's state of operation which is fully synchronized [15] with that of the plant.

The methods developed in this paper use the SCT terminology; interested readers may refer to definitions given in Appendix A.

### 2.3 Supervisory Controller Synthesis

The objective of SCT is to synthesize a supervisor in such a way that the *supervised plant* behaves in accordance with constraints pertaining its restricted behavior. The control specifications provide the constraints to enforce the restricted behavior of the plant. The following steps delineate the synthesis of a supervisory controller (see Section 4 for an illustration of this process; see Appendix A for the definitions of terms in italics referred to in the following steps):

- Model the unsupervised, i.e., open loop physical plant as a DFSA $\mathcal{G}$.
- Provide the specifications of the constrained behavior of $\mathcal{G}$ as English statements. Let $K$ be the formal language obtained for these specifications. Design another DFSA, say $S$, with the same event alphabet $\Sigma$, to produce the language $K$.
- Perform the *completion* on $S$ of the specification language $K$ to obtain the automaton $\overline{S}$ containing the dump state.
- Perform *synchronous composition* ($\mathcal{G} \| \overline{S}$).
- The result of the previous operation is used to verify if the specification given by the language $K \subseteq \Sigma^*$ is *controllable*. The *controllability check* [1], [13] is done to ascertain the existence of a supervisor for the given specification.
- If language $K$ is not controllable, then we have to determine the *supremal controllable sublanguage* $K^{\uparrow C}$ [1], [13]. The resultant automaton for this language is a desired supervisory controller that satisfies the control specification.

In the following sections, we illustrate the synthesis of two controllers for the OS and process interactions.

## 3 Modeling Operating System—Process Interactions

A process is a program in execution. The terms process (i.e., a computer program in execution) and software application are used interchangeably hereafter. For the controller synthesis (see Section 2.3), this section models the interaction between a

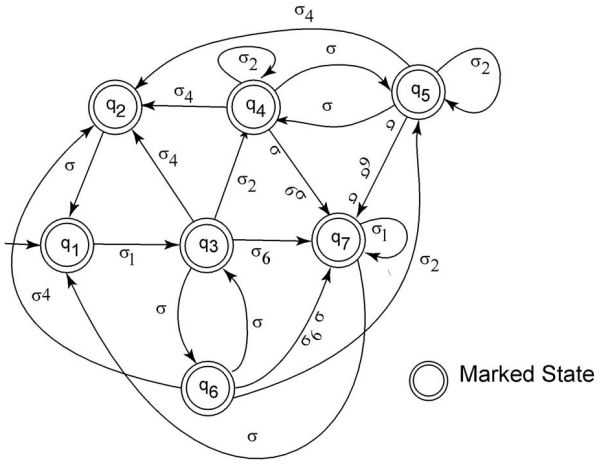



Fig. 1. Process-OS interactions modeled by the DFSA $\mathcal{G}$.

process and the OS as a DFSA plant model that is to be controlled by another DFSA, known as the supervisor DFSA.

### 3.1 Plant Model of OS—Process Interactions

The states in the DFSA model represent operational states of a process while the arcs illustrate the system events leading to transitions between these operational states as shown in Fig. 1. This plant model DFSA is given as a 5-tuple $\mathcal{G} = (Q, \Sigma, \delta, q_1, Q_m)$with initial state as $q_1$. Here, $Q = \{q_1, \ldots, q_7\}$, $\Sigma = \{\sigma_1, \ldots, \sigma_{10}\}$, and $Q_m = Q$ is the set of marked states that represent the completion of important operational states of an application program from an OS perspective (see Fig. 1). Note that $L(\mathcal{G})$ contains only the *legal* (physically admissible) strings that can be achieved starting from the initial state $q_1$. We make $\mathcal{G}$ trim (see Definition A.6) so that it should contain only those states that can be accessed starting from the initial state. By Definitions A.1 and A.2, the language $L(\mathcal{G})$ and marked language $L_m(\mathcal{G}) \subseteq L(\mathcal{G})$ of the DFSA $\mathcal{G}$ are derived as: $L(\mathcal{G}) = \{s \in \Sigma^* | \delta(q_1, s) \in Q\} \subseteq \Sigma^*$ and $L_m(\mathcal{G}) = \{s \in \Sigma^* | \delta(q_1, s) \in Q_m\} \subseteq L(\mathcal{G}) \subseteq \Sigma^*$, where $L_m(\mathcal{G})$ contains all event strings that terminate in a marked state. By Definition A.3, it can be seen that $L(\mathcal{G})$ is prefix-closed, i.e., $L(\mathcal{G}) = pr(L(\mathcal{G}))$.

In this paper, all events in $\Sigma$ that are used to model the automaton $\mathcal{G}$ are assumed to be observable [13] events, i.e., the events are visible to the supervisor. These events are constructed by observing *signals* received by a process [16] and by monitoring the free physical memory resource available to the system. As given in Section 2, we partition the event alphabet $\Sigma$ into subsets of controllable events $\Sigma_c$ and uncontrollable events $\Sigma_{uc}$ such that $\Sigma_{uc} \cup \Sigma_c = \Sigma$ and $\Sigma_{uc} \cap \Sigma_c = \emptyset$. A supervisor controls the computer process by selectively disabling the controllable events based on the control specifications. For a given DFSA plant model $\mathcal{G}$, we define the control specifications that, in turn, generates the supervisor DFSA $S$.

The states and events of a process $\mathcal{G}$ (with initial state $q_1$) are listed in Table 2 and Table 1, respectively. Fig. 1 presents the state transition diagram for the DFSA $\mathcal{G}$ that captures the behavior of a program in execution as it interacts with the OS. Each of the circles with labels represents a state of

TABLE 1
List of Events in DFSA $\mathcal{G}$

| Event | Event Description | Type |
|---|---|---|
| $\sigma_1$ | Start a program | $C$ |
| $\sigma_2$ | CPU generated exception due to program error, non-critical hardware failure. | $UC$ |
| $\sigma_3$ | Process terminated by OS. | $C$ |
| $\sigma_4$ | Execution completed. | $UC$ |
| $\sigma_5$ | Low on resources. | $UC$ |
| $\sigma_6$ | Process terminated (by an external agent.) | $UC$ |
| $\sigma_7$ | Process halted due to lack of resources. | $C$ |
| $\sigma_8$ | Resources available. | $C$ |
| $\sigma_9$ | Process cleanup upon termination | $UC$ |
| $\sigma_{10}$ | Process still resident. | $UC$ |

$C =$ *Controllable Event*, $UC =$ *Uncontrollable Event.*

TABLE 2
List of States $Q$ in DFSA $\mathcal{G}$

| State | State Description |
|---|---|
| $q_1$ | Idle state (Ready to execute) |
| $q_2$ | Execution Completed |
| $q_3$ | Normal Execution |
| $q_4$ | Process Fault Detected |
| $q_5$ | Possible faulty execution as well as low on resources |
| $q_6$ | Normal execution but low on resources (DP) |
| $q_7$ | Process Halted |

the DFSA $\mathcal{G}$. Labels on the arcs are the events contained in the alphabet $\Sigma$ of the automaton $\mathcal{G}$.

#### 3.1.1 The Plant DFSA Model

The initial state $q_1$ in Fig. 1 is the idle state in which the OS is ready to execute a new process. The program start (event $\sigma_1$) is the only event that can produce a transition from $q_1$. The system is in state $q_3$ under normal execution of the program. There are four possible transitions from $q_3$ in the unsupervised model. The process can remain in $q_3$ during its lifetime if it does not incur any exceptions and then move to $q_2$ by exiting on its own volition using the system call `exit()` or when the program control flow reaches the last statement of the main procedure `main()`. However, if the program causes the CPU to generate an exception due to a program error (event $\sigma_2$), its state changes to $q_4$, where the OS executes its exception handler in the context of the process and then notifies the process by a *signal* of the anomalous condition. For a fatal exception, the OS takes the default action of terminating the process even if a *signal handler* [16] is provided. We model this behavior as an event $\sigma_3$ arising from state $q_3$. For a nonfatal exception, the signal handler (if provided) is executed and the process (if not terminated by the OS) will continue to run at state $q_4$.

At state $q_3$, it is possible that the process is terminated by the user which is shown by the event $\sigma_6$ that causes a transition to state $q_7$. The user's decision to terminate the process cannot be controlled and, therefore, the event $\sigma_6$ is made uncontrollable. Such an event is possible from other states of the task and, hence, we see $\sigma_6$ from all states except $q_1$. The self-loop of event $\sigma_{10}$ indicates that the OS has not yet recovered all the resources locked by the process. Event $\sigma_9$ represents the actions of the OS to release any

resources owned by the process. These resources include memory, open files, and possibly synchronization resources such as semaphores. Any process that completes its execution via event $\sigma_4$ traverses to $q_2$ and thereafter to state $q_1$ via $\sigma_9$.

States $q_5$ and $q_6$ depict the scenario in which the system is low on resources such as memory. While $q_6$ can be reached from $q_3$ via the low resource event $\sigma_5$, $q_5$ can be reached from $q_4$ and $q_6$ via events $\sigma_5$ and $\sigma_2$, respectively. By giving state $q_5$, we want to show that a process can incur exceptions when running with inadequate resources. Observe the self-loop at $q_4$ due to the program error event $\sigma_2$ that implies that a task can execute in state $q_4$ while causing exceptions.

When the system is low on resources (e.g., memory leaks are one of the reasons causing the free physical memory resource to reduce over time and are observed as the running software ages [4]), the OS may take actions to create resources for normal operation of the task and this is accomplished by using the event $\sigma_8$. It is also possible that the OS does not take any such action and the task is allowed to execute in the same state where there are chances that it can cause exceptions as shown by the $\sigma_2$ self-loop at $q_5$. The OS may halt a task if it has insufficient resources for further execution, in which case $\sigma_7$ causes transition to $q_7$. As long as the process resources remain locked, the system is at $q_7$ due to the self-loop $\sigma_{10}$. We do not consider a resumption of the process from state $q_7$. Therefore, as and when the OS releases these resources after it has removed the process's descriptor from memory, event $\sigma_9$ captures this behavior by producing a transition to the idle state $q_1$.

### 3.2 Marked States and Weight Assignments

The purpose of making a state "marked" is that the state should represent completed or important operational phases of the physical plant represented by the model. In the plant model, we have made all the states as marked because of their importance during the execution of the process. The marked language of the plant initialized at the state $q_1$ is given by $L_m(\mathcal{G}) = \{s \in \Sigma^* | \delta^*(q_1, s) \in Q_m\}$. In order to obtain a quantitative measure of $L_m(\mathcal{G})$, we partition the plant's set $Q_m$ of marked states into subsets of good and bad marked states. A good marked state is assigned a positive value while a bad marked state is given a negative value. We characterize each marked state by assigning a signed real value on the scale of $[-1, 1]$ based on our perception of the state's impact on the performance of the software system. A lower value assigned to a marked state implies a greater degradation in the performance of the system in that state.

Marked states of the plant $\mathcal{G}$ are assigned the weights given by the characteristic vector

$$\overline{\chi_{\mathcal{G}}} = [0.3\ 0.8\ 0.8\ 0.1\ 0.08\ 0.4\ -0.8]^T.$$

For example, in state $q_7$, we consider that the software application has failed and cannot recover. State $q_7$ is therefore assigned a high negative weight of -0.8 but not -1 as it does not necessarily represent a failure of the OS and other applications. State $q_1$ has a weight of 0.3 since the application has yet to enter the existing workload on the system, so we give $q_1$ a relatively lower weight. Together, states $q_2$ and $q_3$ represent the best desired operation of the system where a process assigned to the CPU for execution, performs its operation and relinquishes control upon completion without producing conditions that adversely affect the system performance. Hence, they have weights of 0.8 each. Software faults at state $q_4$ can cause the software to produce erroneous results, we consider this to be a degraded performance of the system and assign it a comparatively lower weight of 0.1. In state $q_5$, the system is in a condition that is relatively worse than that in state $q_4$ due to depletion of resources, e.g., free physical memory, which is an additional reason for a degraded execution. Therefore, it has a weight of 0.08, which is lower than that of state $q_4$. With no prior faulty operation except for reduced resource levels in state $q_6$, we consider the system to be in an operational state that is better than states $q_4$ and $q_5$. So, it is assigned a weight of 0.4, which is higher than the weights of both $q_4$ and $q_5$.

## 4 Formulation of Supervisory Control Policies

This section devises two control policies under supervisors $S_1$ and $S_2$. The specifications of these two control policies are as follows:

Policy 1 ($S_1$). Prevent termination of process on first occurrence of a fatal exception. Enable termination of process on the second occurrence of this exception. Prevent halting of process due to low resources.

Policy 2 ($S_2$). Prevent termination of process on first and second occurrence of a fatal exception. Enable termination of process on the third occurrence of this exception. Prevent halting of process due to low resources.

Let $K_1$ and $K_2$ be the regular languages induced by the specifications of the supervisors $S_1$ and $S_2$ that are synthesized separately in the following two subsections. The languages $K_1$ and $K_2$ are regular and are therefore converted into the respective supervisor's DFSA [13], [6] as seen in Fig. 2a and Fig. 2b. The dotted lines signify that the associated controllable events are disabled according to the specifications. Note that the event alphabet $\Sigma$ is common to the plant model and both the supervisors. In addition, it is assumed that all events are observable (see Section 3). However, there are no restrictions on the state set $X$ (see (2)) to be same as the state set $Q$ of the plant model. Table 3 lists the states of the supervisory control automata generated from the specification languages $K_1$ and $K_2$.

### 4.1 Synthesis of Supervisor $S_1$

Let $S_1$ be the DFSA (see Fig. 2a) created for the specification given by Policy 1. Let $K_1$ be the language of this supervisor DFSA. Fig. 2a shows that there are additional states in the supervisor automaton than in plant model automaton $\mathcal{G}$.

We analyze the supervisor DFSA given in Fig. 2a to examine the restrictions enforced on the plant's behavior by the specification $S_1$ (Policy 1). The supervisor captures the start of a process through event $\sigma_1$ which produces a transition from state $x_1$ to $x_3$. The first time a process causes an exception, the supervisor observes it through the transition from state $x_3$ to state $x_4$. The process is not

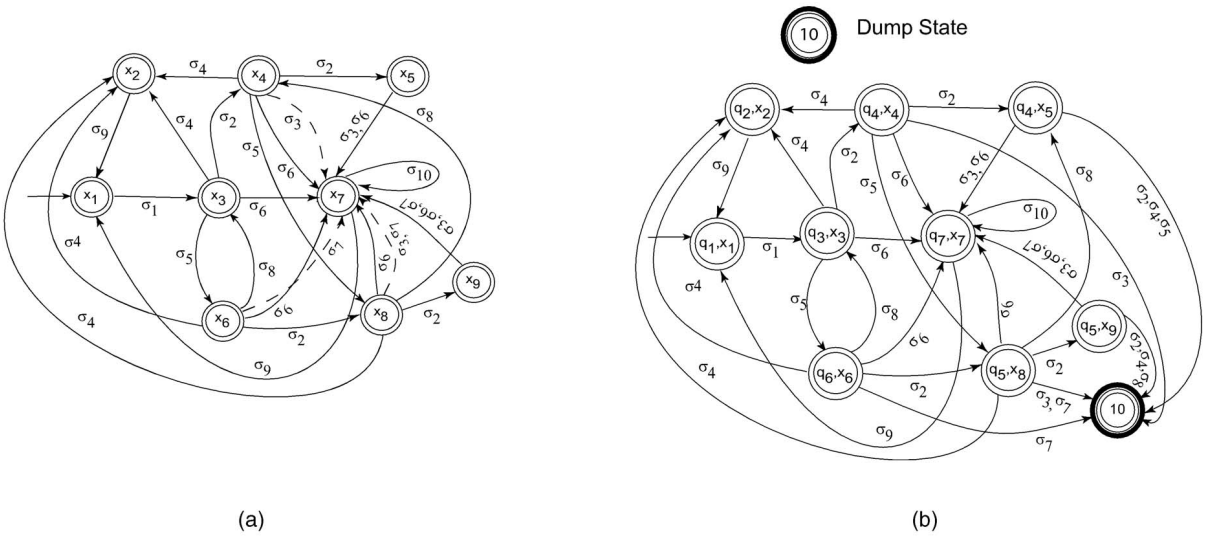



Fig. 2. (a) and (b) show the DFSA of the sublanguage $K_1$ for supervisor $S_1$ and its failure to satisfy the *Controllability Condition*, respectively. Dashed lines show controllable events disabled at states as per the specification given by Policy 1. (a) DFSA of the control specification language $K_1$ for supervisor $S_1$ obtained from the specification given by Policy 1. (b) DFSA of $(\mathcal{G}\|\overline{S_1})$ is the *Completion* of automaton $S_1$ given in (a).

TABLE 3
List of States in DFSAs $S_1$ and $S_2$

| $S_1$'s States | $S_2$'s States | State Description |
|---|---|---|
| $x_1$ | $x_1$ | Idle state (Ready to execute) |
| $x_2$ | $x_2$ | Execution Completed |
| $x_3$ | $x_3$ | Normal Execution |
| $x_4$ | $x_4, x_5$ | Process Fault Detected. Permit further execution |
| $x_5$ | $x_9$ | Process Fault Detected. Permit process termination due to fault |
| $x_8$ | $x_8, x_{10}$ | Low on resources. Permit execution inspite of faults |
| $x_9$ | $x_{11}$ | Low on resources. Permit process termination due to fault |
| $x_6$ | $x_6$ | Normal execution but low on resources (DP) |
| $x_7$ | $x_7$ | Process Halted |

terminated immediately as we disable the event $\sigma_3$ at state $x_4$, i.e., prevent the OS from terminating the process.

In $\mathcal{G}$ at state $q_4$ (see Fig. 1), we show a self-loop due to the additional occurrences of exceptions while running in that state. The specification of $S_1$ restricts the language of $\mathcal{G}$ such that an event sequence it produces contains at the most two instances of the event $\sigma_2$. Therefore, to remove the effect of the self-loop, we create state $x_5$ to which the supervisor moves on the second occurrence of an exception, i.e., the event $\sigma_2$. Note that $x_4$ and $x_5$ are states where the process is not low on a resource such as memory.

Similar counting of exceptions is performed when the process shows a degraded state of operation due to less than required resources. The supervisor has state $x_6$ to indicate that the process is running on low physical memory. As modeled in $\mathcal{G}$, under low resource conditions, the process moves to state $q_5$ when an exception occurs and continues there if additional exceptions occur (shown by the self-loop at state $q_5$ in Fig. 1). Under this situation, the supervisor is required to restrict the language produced by the process and, so, states $x_8$ and $x_9$ are added in the supervisor automaton. The policy states that the process should not halt due to less available memory and, so, we disable the event $\sigma_7$ from states $x_6$ and $x_8$. Enabling $\sigma_7$ at state $x_9$ does not affect the specification as the OS terminates the process when event $\sigma_3$ is enabled at state $x_9$.

#### 4.1.1 Controllability of Supervisor $S_1$

In order to determine if the specification language $K_1$ is controllable we verify if the *controllability condition* (see Definition A.9, [1]) on $K_1$ is satisfied. For this we create the DFSA $(\mathcal{G}\|\overline{S_1})$ (see Fig. 2b) by the synchronous composition operation (see Definition A.7) of $\mathcal{G}$ and $\overline{S_1}$, where $\overline{S_1}$ is the *completion* of the automaton $S_1$ in Fig. 2a, with dump state. In Fig. 2b, state "10" is the *dump state*.

$K_1 \subseteq L(\mathcal{G})$ is controllable if and only if, for each $s \in pr(K_1)$ and $u \in \Sigma_{uc}$ such that $su \in L(\mathcal{G})$, then we have $su \in pr(K_1)$. Fig. 2b shows that there are uncontrollable events from states $(q_4, x_5)$ and $(q_5, x_9)$ leading to the dump state "10.".Observe that a string in Fig. 2b such as $(\sigma_1\sigma_5\sigma_8\sigma_2\sigma_2\sigma_5) \notin pr(K_1)$. The supervisor therefore does not enable feasible uncontrollable events possible in the plant. Thus, the controllability condition on $K_1$ is not satisfied and, so, $K_1$ is not controllable.

Since the controllability condition does not hold on $K_1$, we determine the *supremal controllable sublanguage* (see

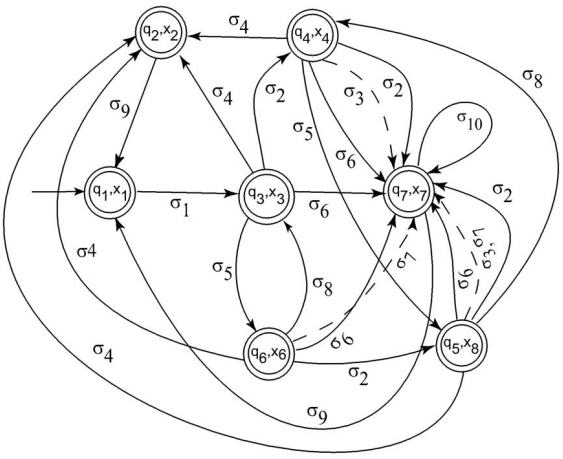


Fig. 3. DFSA for Supervised process $(S_1/\mathcal{G})$ using the supremal controllable sublanguage $K_1^{\uparrow C}$ for supervisor $S_1$ induced by the specification Policy 1. Dashed lines show the controllable events that are disabled at the states.

Definition A.10 and [13]) of $K_1$ . To obtain the supremal controllable sublanguage $K_1^{\uparrow C}$ of $K_1$, we remove the strings with uncontrollable prefixes, e.g., $(\sigma_1\sigma_5\sigma_8\sigma_2\sigma_2)$. To remove the set of strings with uncontrollable prefixes, it suffices to remove the states reached by their execution in $(\mathcal{G}\|\overline{S_1})$. Thus, we remove the state $(q_4, x_5)$, $(q_5, x_9)$ and the *dump state*—"10" from $(\mathcal{G}\|\overline{S_1})$. The result of this operation is shown in Fig. 3. The DFSA shown in Fig. 3 is our controller $(S_1/\mathcal{G})$ that satisfies the control policy (Policy 1) in the supervisor $S_1$.

#### 4.1.2 Weight Assignment to Marked States

The weights assigned assigned to the marked states of the supervisor $S_1/\mathcal{G}$ are given by the characteristic vector $\overline{\chi}_{S_1/\mathcal{G}} = [0.3 \quad 0.8 \quad 0.8 \quad 0.1 \quad 0.08 \quad 0.4 \quad -0.8]^T$. We apply the same rationale for assigning weights as done for the plant $\mathcal{G}$ in Section 3.2.

### 4.2 Synthesis of Supervisor $S_2$

Let $S_2$ be the supervisor DFSA for the specification given by Policy 2 based on the language $K_2$, as shown in Fig. 4a. The supervisor $S_2$ is similar to the supervisor $S_1$ with the difference that it allows the process to execute even after the second occurrence of an exception.

The supervisor $S_2$ differs from supervisor $S_1$ in the number of exceptions it permits before the OS is allowed to terminate the application. Similar to supervisor $S_1$, counting is made possible by adding states. Fig. 4a shows that we add states $x_5$ and $x_9$ when the system is not low on resources and states $x_{10}$, $x_{11}$ to satisfy the control policy.

#### 4.2.1 Controllability of Supervisor $S_2$

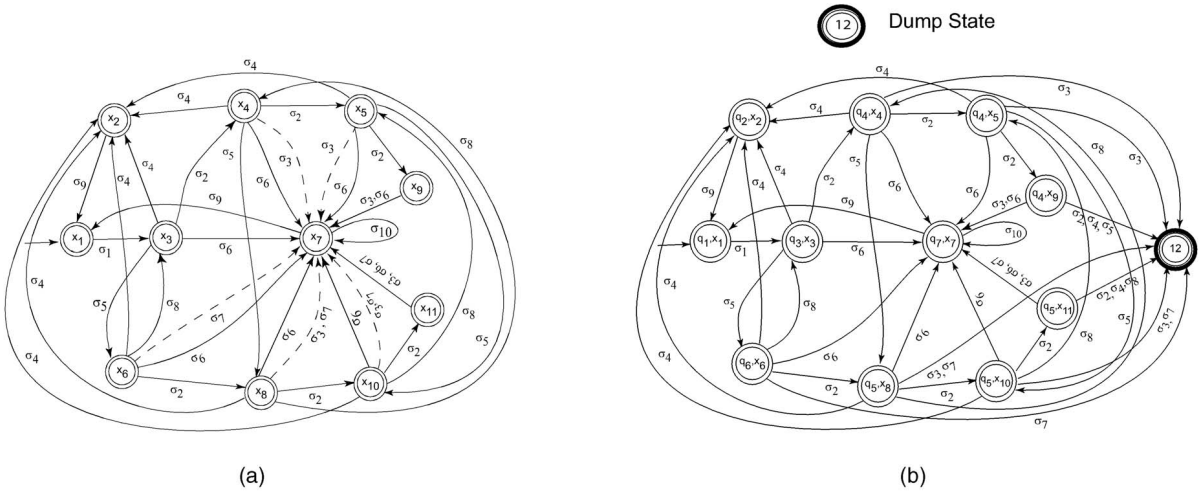
Fig. 4b shows that specification $K_2$ as there are uncontrollable events leading to the dump state. Therefore, we determine the supremal controllable sublanguage $K_2^{\uparrow C}$ as in the case of supervisor $S_1$. Fig. 5 shows the automaton for the desired controller $(S_2/\mathcal{G})$ that satisfies Policy (Policy 2).

#### 4.2.2 Weight Assignment to Marked States

The weights assigned assigned to the marked states of the supervisor $S_2/\mathcal{G}$ are given by the characteristic vector $\overline{\chi}_{S_2/\mathcal{G}} = [0.3 \quad 0.8 \quad 0.8 \quad 0.1 \quad 0.1 \quad 0.08 \quad 0.08 \quad 0.4 \quad -0.8]^T$. The same rationale as for the plant $\mathcal{G}$ is applied for the weight assignment here. New states that are created are assigned weights depending on the state of the plant with which the supervisor state has combined. Therefore, states $(q_4, x_5)$ and $(q_5, x_{10})$ are assigned weights of 0.1 and 0.08, respectively.

## 5 Implementation of Supervisory Control

This section presents our software architectural framework for realizing control in software systems. Fig. 6 shows the supervisory control system comprised of four separate components in labeled ellipses above the rectangular box and representing individual user level processes. However, only a combination of the three fundamental processes




(a) (b)

Fig. 4. (a) and (b) show the DFSA of the sublanguage $K_2$ for supervisor $S_2$ and its failure to satisfy the *Controllability Condition*, respectively. Dashed lines show controllable events disabled at states as per the specification given by Policy 2. (a) DFSA of the control specification $K_2$ for supervisor $S_2$ obtained from the specification given by Policy 2. (b) DFSA of $\mathcal{G}\|\overline{S}_2$ is the *Completion* of $S_2$ automaton given in (a).

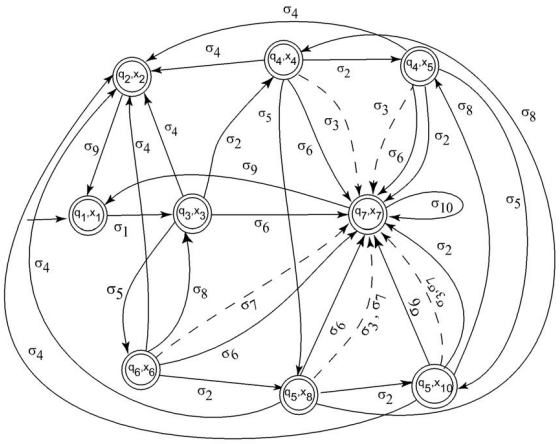


Fig. 5. DFSA for Supervise process $S_2/\mathcal{G}$ using the supremal controllable sublanguage $K_2^{\uparrow C}$ for supervisor $S_2$ given by the specification of Policy 2. Dashed lines show the controllable events that are disabled at the states.

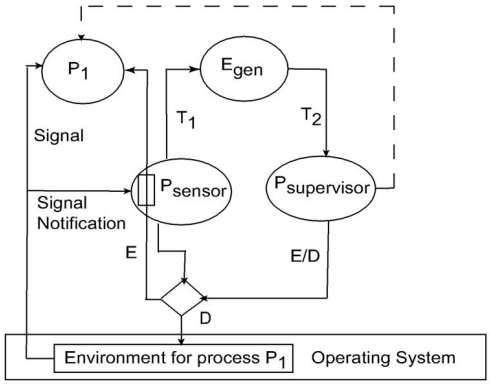



Fig. 6. Framework for supervisory control of OS.

$P_{sensor}$, $E_{gen}$, $P_{supervisor}$ exercise control over the OS. As we intend to achieve control over an application indirectly by being able to access the OS, the process represented by the ellipse labeled $P_1$ is indirectly controlled by the supervisory control system. The functional processes $P_{sensor}$, $E_{gen}$, $P_{supervisor}$ in Fig. 6 are implemented as user level processes with the capability to insert *kernel modules* [17] so that the control path can be changed to satisfy the supervisory control policy. Entire implementation of the supervisory control system is under the Red Hat Linux 7.2 OS [16]. For simplicity of illustration, this paper shows the implementation for a single process under supervisory control.

## 5.1 Components of the SCT Framework

The components of the supervisory control system are the following:

1. $P_{sensor}$—The sensor/actuator system has two functions:
   a. monitoring signals and collection of resource status information received from the OS and
   b. implementation of supervisory decisions.
2. $E_{gen}$—The event generator maps the resource measurements and signals received by the process into corresponding higher-level events.
3. $P_{supervisor}$—The supervisor automaton is designed based on the control specification.

### 5.1.1 The Sensor/Actuator System

The sensor system implements two threads to track a process. The threads perform 1) signal capturing and 2) process information collection, respectively. To intercept the signals received by the process the sensor uses `ptrace()` and `wait()` [16] system calls. We use the Linux */proc* file system (indexed by the process identifier) to obtain relevant system information of process $P_1$. The information pertains to the computer process parameters (e.g., code size and memory usage) and is sent as a different message type to the event generator. The event generator uses this data for the generation of a high level events.

### 5.1.2 The Event Generator

The supervisor process that executes the controlled language automaton $S/\mathcal{G}$ can perceive only high level events (see Table 1). $E_{gen}$ maps sensor signals and the data provided by the process information collection thread as higher level events compatible with $P_{supervisor}$. $E_{gen}$ implements both one-one mapping as well as the many-one mapping of signals to events. Mapping of signals and resource information to high level events is shown in Table 4.

### 5.1.3 The Supervisor Automaton

The controlled language automaton (e.g., $S_1/\mathcal{G}$ and $S_2/\mathcal{G}$ ) is executed by the supervisor process. The supervisor is implemented to control a process using the information provided by $E_{gen}$ as a message containing the event through the communication link $T_2$. The supervisor dynamically makes decisions of disabling certain controllable events depending on its current state. The status of each controllable event—*enabled* $(E)$ or *disabled* $(D)$ is then sent to $P_{sensor}$ via the protocol $E/D$. The sensor process also performs the task of an actuator to enforce supervisor's decisions that may cause the actions of the OS to change. In the end, the results are reflected in the actions taken by the OS on the application program tracked by the sensor.

## 5.2 Operation of SCT Framework

RedHat Linux 7.2 is a multitasking operating system. Here, a kernel maintains a status repository for each process. The kernel keeps track of information such as whether a particular process is currently executing on the CPU or whether it is blocked on an event, the address space assigned to the process, and the files the process has opened during its life time, etc. To facilitate the maintenance of this information, a process descriptor is implemented in the kernel using the data structure `task_struct`. Fig. 6 shows a rectangular box labeled environment of the process $P_1$ which represents the information base for this process.

We use BSD sockets [18] to communicate between different components of our system. Different communication protocols that we have developed (see Fig. 6) are: Protocol $T_1$ shown by the unidirectional arrow from $P_{sensor}$ to $E_{gen}$. Protocol $T_2$ shown by the unidirectional arrow from $E_{gen}$ to $P_{supervisor}$. Protocol $E/D$ shown by the arrow going from $P_{supervisor}$ to $P_{sensor}$. $P_{sensor}$ uses the protocol $T_1$ to send

TABLE 4
Mapping Signals and Data Collected for /proc Filesystem into Events of the Plant $\mathcal{G}$

| Event | Mapping Conditions |
|---|---|
| $\sigma_1$ | `SIGSTOP` sent to process when sensor uses the `ptrace()` system call. |
| $\sigma_2$ | `wait()` system call's `WSTOPSIG` macro returns the signal numbers `SIGFPE` or `SIGSEGV`. |
| $\sigma_3$ | `wait()` system call's `WIFSIGNALED` macro returns true but termination not due to `SIGKILL` signal (`WTERMSIG` $\neq$ `SIGKILL`). |
| $\sigma_4$ | `wait()` system call's `WIFEXITED` macro evaluates to a value $> 0$. |
| $\sigma_5$ | Free memory available is below user defined threshold (FM $< 5\%$ of PM). |
| $\sigma_6$ | `SIGKILL` detected by `wait()` system call's `WTERMSIG` macro. |
| $\sigma_7$ | Free memory available below user defined threshold (FM $< 1\%$ of PM). |
| $\sigma_8$ | Available free memory above user defined threshold (FM $> 5\%$ of PM). |
| $\sigma_9$ | `/proc` contains no entry of process ID (PID). |
| $\sigma_{10}$ | `/proc` contains process information after `SIGKILL` or `WTERMSIG` $\neq$ `SIGKILL`. |

*PM = Total Physical Memory, FM = Available Free Physical Memory.*

the raw signal information it collects to $E_{gen}$. The event generator creates events from these raw signals and then, using $T_2$, sends this information to the supervisor process. The supervisor process executing the $S/\mathcal{G}$ automaton is only capable of disabling or enabling the events and, hence, we provide the $E/D$ (*enable/disable*) decision that the $P_{supervisor}$ sends to the sensor depending on the current state of the automaton $S/\mathcal{G}$. The sensor process has control over the process, but we wish to control the process by making the operating system deviate from its normal control path.

The sensor process has the ability to dynamically load and unload *kernel modules* [17] to create the deviations from the original control path. These deviations are needed when a control policy designates a particular event to be disabled in a particular state. This action is shown by the arrow labeled $D$ emanating from the decision box. For an enabled event, the process should be made aware of the situation and, hence, we show the arrow labeled $E$ going to the process. Fig. 6 also shows a set of arrows emanating from the environment of the process. The labels *Signal* and *Signal Notification* relates directly to the mechanism the `ptrace()` system call uses for making the sensor process aware of a signal (e.g., `SIGSTOP`, `SIGWINCH`, `SIGFPE`, `SIGSEGV`, etc.) a process has received.

## 6 Performance Measurement of Controllers

In general, the controlled sublanguages of a plant language could be different under different controllers that satisfy their respective policies. The partially ordered set of sublanguages can then be totally ordered using their respective performance measures [10], [11], [12]. This total ordering using performance measures facilitates comparison between the unsupervised and the supervised plant models, therefore, we apply the language measure theory to compute the performance measures of the DFSAs we have designed. Comparisons of the models along with the corresponding performance results are shown in Section 8.2. The theoretical foundation of this performance measure and its mathematical interpretation is given below.

### 6.1 Language Measure Theory

This section briefly reviews the concept of signed real measure of regular languages [10], [11], [12]. Consider a plant behavior be modeled as a DFSA $G_i \equiv (Q, \Sigma, \delta, q_i, Q_m)$ as given in (1) and, therefore, $Q$, $\delta$, and $Q_m$ have the same meaning. Furthermore, we consider $|Q| = n$ excluding the dump state [1], if any, and $|\Sigma| = m$. To represent the languages of strings starting from each state $q \in Q$, we consider each such state as $\forall i \in \mathcal{I} \equiv \{0, \ldots, n-1\}\ \forall q_i \in Q$ at which the DFSA is initialized. Then, the languages generated by such a DFSA when initialized at a state $q_i$ are a generalization of Definitions A.1 and A.2 and are given below.

**Definition 1.** *A DFSA $G_i$, initialized at $q_i \in Q$, generates the language $L(G_i) \equiv \{s \in \Sigma^* : \delta^*(q_i, s) \in Q\}$ and its marked sublanguage $L_m(G_i) \equiv \{s \in \Sigma^* : \delta^*(q_i, s) \in Q_m\}$.*

The language $L(G_i)$ is partitioned as the nonmarked and the marked languages, $L^o(G_i) \equiv L(G_i) - L_m(G_i)$ and $L_m(G_i)$, consisting of event strings that, starting from $q \in Q$, terminate at one of the nonmarked states in $Q - Q_m$ and one of the marked states in $Q_m$, respectively. The set $Q_m$ is partitioned into $Q_m^+$ and $Q_m^-$, where $Q_m^+$ contains all *good* marked states that we desire to reach and $Q_m^-$ contains all *bad* marked states that we want to avoid, although it may not always be possible to avoid the bad states while attempting to reach the good states. The marked language $L_m(G_i)$ is further partitioned into $L_m^+(G_i)$ and $L_m^-(G_i)$ consisting of good and bad strings that, starting from $q_i$, terminate on $Q_m^+$ and $Q_m^-$, respectively.

A signed real measure $\mu : 2^{\Sigma^*} \rightarrow \Re \equiv (-\infty, \infty)$ is constructed for quantitative evaluation of every event string $s \in \Sigma^*$. The language $L(G_i)$ is decomposed into null (i.e., $L^o(G_i)$), positive (i.e., $L_m^+(G_i)$), and negative (i.e., $L_m^-(G_i)$) sublanguages.

**Definition 2.** *The language of all strings that, starting at a state $q_i \in Q$, terminates on a state $q_j \in Q$, is denoted as $L(q_i, q_j)$. That is, $L(q_i, q_j) \equiv \{s \in L(G_i) : \delta^*(q_i, s) = q_j\}$.*

**Definition 3.** *The characteristic function that assigns a signed real weight to state-partitioned sublanguages $L(q_i, q_j)$, $i = 0, 1, \ldots, n-1$, is defined as: $\chi : Q \rightarrow [-1,\ 1]$ such that*

$$\chi(q_j) \in \begin{cases} [-1,\,0)\ if\ q_j \in Q_m^- \\ \{0\} \quad if\ q_j \notin Q_m \\ (0,\,1]\ if\ q_j \in Q_m^+. \end{cases}$$

**Definition 4.** *The event cost is conditioned on a DFSA state at which the event is generated, and is defined as* $\tilde{\pi}: \Sigma^* \times Q \to [0,\,1]$ *such that* $\forall q_j \in Q,\ \forall \sigma_k \in \Sigma,\ \forall s \in \Sigma^*$, $\tilde{\pi}(\sigma_k,\,q_j) = 0$ *if* $\delta(q_j, \sigma_k)$ *is undefined;* $\tilde{\pi}(\varepsilon,\,q_j) = 1$, $\tilde{\pi}(\sigma_k,\,q_j) \equiv \tilde{\pi}_{jk} \in [0,\,1)$, $\sum_k \tilde{\pi}_{jk} < 1$

$$\tilde{\pi}(\sigma_k\, s,\,q_j) = \tilde{\pi}(\sigma_k,\,q_j)\,\tilde{\pi}(s, \delta(q_j, \sigma_k)).$$

*The event cost matrix, denoted as* $\tilde{\Pi}$-*matrix, is defined as:*

$$\tilde{\Pi} = \begin{bmatrix} \tilde{\pi}_{00} & \tilde{\pi}_{01} & \cdots & \tilde{\pi}_{0m-1} \\ \tilde{\pi}_{10} & \tilde{\pi}_{11} & \cdots & \tilde{\pi}_{1m-1} \\ \vdots & & \ddots & \vdots \\ \tilde{\pi}_{n-10} & \tilde{\pi}_{n1} & \cdots & \tilde{\pi}_{n-1m-1}. \end{bmatrix}$$

Now we define the measure of a sublanguage of the plant language $L(G_i)$ in terms of the signed characteristic function $\chi$ and the nonnegative event cost $\tilde{\pi}$.

**Definition 5.** *Given a DFSA* $G_i \equiv \langle Q, \Sigma, \delta, q_i, Q_m \rangle$, *the cost* $\nu$ *of a sublanguage* $K \subseteq L(G_i)$ *is defined as the sum of the event cost* $\tilde{\pi}$ *of individual strings belonging to* $K$ $\nu(K) = \sum_{s \in K} \tilde{\pi}(s, q_i)$.

**Definition 6.** *The signed real measure* $\mu$ *of a singleton string set* $\{s\} \subset L(q_i, q_j) \subseteq L(G_i) \in 2^{\Sigma^*}$ *is defined as:* $\mu(\{s\}) \equiv \chi(q_j)\,\tilde{\pi}(s,\,q_i) \quad \forall s \in L(q_i, q_j)$.

*The signed real measure of* $L(q_i, q_j)$ *is defined as:* $\mu\big(L(q_i, q_j)\big) \equiv \sum_{s \in L(q_i, q_j)} \mu(\{s\})$ *and the signed real measure of a DFSA* $G_i$, *initialized at the state* $q_i \in Q$, *is denoted as:* $\mu_i \equiv \mu(L(G)) = \sum_j \mu\big(L(q_i, q_j)\big)$.

**Definition 7.** *The state transition cost of the DFSA is defined as a function* $\pi: Q \times Q \to [0,\,1)$ *such that* $\forall q_j,\ q_k \in Q$, $\pi(q_j,\,q_k) = \sum_{\sigma \in \Sigma:\,\delta(q_j,\,\sigma) = q_k} \tilde{\pi}(\sigma,\,q_j) \equiv \pi_{jk}$ *and* $\pi_{jk} = 0$ *if* $\{\sigma \in \Sigma:\ \delta(q_j,\,\sigma) = q_k\} = \emptyset$. *The state transition cost matrix, denoted as* $\Pi$-*matrix, is defined as:*

$$\Pi = \begin{bmatrix} \pi_{00} & \pi_{01} & \cdots & \pi_{0n-1} \\ \pi_{10} & \pi_{11} & \cdots & \pi_{1n-1} \\ \vdots & & \ddots & \vdots \\ \pi_{n-10} & \pi_{n-11} & \cdots & \pi_{n-1n-1} \end{bmatrix}.$$

Wang and Ray [10] have shown that the measure $\mu_i \equiv \mu(L(G_i))$ of the language $L(G_i)$, with the initial state $q_i$, can be expressed as: $\mu_i = \sum_j \pi_{ij}\,\mu_j + \chi_i$, where $\chi_i \equiv \chi(q_i)$. Equivalently, in vector notation: $\bar{\mu} = \Pi\bar{\mu} + \bar{\chi}$, where the measure vector $\bar{\mu} \equiv [\mu_1\ \mu_2\ \cdots\ \mu_n]^T$ and the characteristic vector $\bar{\chi} \equiv [\chi_1\ \chi_2\ \cdots\ \chi_n]^T$.

### 6.2 Probabilistic Interpretation

The signed real measure (Definition 6) for a DFSA is based on the assignment of the characteristic vector and the event cost matrix. As stated earlier, the characteristic function is chosen by the designer based on his/her perception of the states' impact on system performance. On the other hand, the event cost is an intrinsic property of the plant. The event cost $\tilde{\pi}_{jk}$ is conceptually similar to the state-based conditional probability as in Markov Chains, except for the fact that it is not allowed to satisfy the equality condition $\sum_k \tilde{\pi}_{jk} = 1$. Note that $\sum_k \tilde{\pi}_{jk} < 1$ is a sufficiency condition for convergence of the language measure [10], [19].

With this interpretation of event cost, $\tilde{\pi}[s, q_i]$ (Definition 4) denotes the probability of occurrence of the event string $s$ in the plant model $G_i$ starting at state $q_i$ and terminating at state $\delta^*(s, q_i)$. Hence, $\nu(L(q, q_i))$ (Definition 5), which is a nonnegative real number, is directly related (but not necessarily equal) to the total probability that state $q_i$ would be reached as the plant operates. The language measure $\mu_i \equiv \mu(L(G_i)) = \sum_{q \in Q} \mu(L(q_i, q)) = \sum_{q \in Q} \nu(L(q_i, q))\chi(q)$ is then directly related (but not necessarily equal) to the expected value of the characteristic function. Therefore, in the setting of language measure, a supervisor's performance is superior if the supervised plant is more likely to terminate at a *good* marked state and/or less likely to terminate at a *bad* marked state.

## 7 Estimation of Event Cost $\tilde{\Pi}$-Matrix Parameters

We use the result of our derivation of the theoretical bound, $N_b$, on the number of experimental observations (see (6), Appendix B) for each state of a DFSA to compute the event cost matrix parameters for a given $\delta$ and $\varepsilon$. Let $p_{ij}$ be defined as the transition probability of event $\sigma_j$ on the state $q_i$, i.e.,

$$p_{ij} = \begin{cases} P[\sigma_j | q_i], & \text{if } \exists q \in Q,\ s.t.\ q = \delta(q_i, \sigma_j) \\ 0, & \text{otherwise} \end{cases}$$

and its estimate $\hat{p}_{ij}$ that is to be estimated from the ensemble of experiments and/or simulation data. We introduce the indicator function $I_t(i, j)$ to represent the occurrence of event $\sigma_j$ at time $t$ if the system was in state $q_i$ at time $t-1$, where $t$ represents a generalized time epoch, for example, the experiment number. Formally, $I_t(i, j)$ is expressed as:

$$I_t(i, j) = \begin{cases} 1, & \text{if } \sigma_j \text{ is observed at state } q_i \\ 0, & \text{otherwise.} \end{cases}$$

Let $N_t(i)$, denoting the number of incidents of reaching the state $q_i$ up to the time instant $t$, be a random process mapping the time interval up to the instant $t$ into the set of nonnegative integers. Similarly, let $n_t(i, j)$ denote the number of occurrences of the event $\sigma_j$ at the state $q_i$ up to the time instant $t$.

The plant model in general is an inexact representation of the physical plant; therefore, there exist unmodeled dynamics which can manifest themselves either as unmodeled events that may occur at each state or as unaccounted states in the model. Let $\Sigma_i^u$ denote the set of unmodeled events at state $i$ of the DFSA. Therefore, the residue $\theta_i = 1 - \sum_j \tilde{\pi}_{ij}$ denotes the probability of all the unmodeled events emanating state $q_i$. Let

$$\Sigma^u = \cup_i \Sigma_i^u, \forall i \in \mathcal{I} \equiv \{1, \ldots, n\},$$

at each state $q_i$, $P[\Sigma^u | q_i] = \theta_i \in (0, 1)$ and $\sum_i \tilde{\pi}_{ij} = 1 - \theta_i$. Therefore, an estimate of the $(i, j)$th element in $\tilde{\boldsymbol{\Pi}}$-matrix, denoted $\hat{\tilde{\pi}}_{ij}$, is obtained by

$$\hat{\tilde{\pi}}_{ij} = \hat{p}_{ij}(1 - \theta_i). \tag{3}$$

Since $\theta_i \ll 1$, an alternative approximation approach is taken for ease of implementation. We set $\theta_i = \theta \; \forall i$, where the parameter $0 < \theta \ll 1$ is selected from the numerical perspective based on the fact that the sup-norm $\|\mu\|_\infty \leq \theta^{-1}$ [10], [12].

Following is the formulation of the Event Cost Matrix Parameter Estimation algorithm we use to compute elements of $\tilde{\Pi}$-matrix.

**Algorithm: Event Cost Matrix Parameter Estimation**

(1) Initialize $\forall q_i \in Q \; n_0(i) = 0$ and $N_0(i) = 0$

(2) Compute $N$ for a given $\delta$ and $\varepsilon$ using (6) (see Appendix B)

```
/*For each state q_i ∈ Q check if it occurs in
the tth experiment. If a state q_i occurs then
increment its occurrence count. Similarly
if an event σ_j occurs at state q_i in the tth
experiment then increment the event
occurrence count for event σ_j at state q_i In
order to obtain stable state transition
probabilities this loop is repeated until
all states reach the upper bound computed in
step 2.
*/
```

(3)
```
do
   for i=1 to |Q|
      N_t(i) = N_{t-1}(i) + I_t(i)
      n_t(i,j) = n_{t-1}(i,j) + I_t(i,j)
   end
until ∀ q_i ∈ Q, min{N_t(i)} ≤ N

/*Each element π̂̃_ij of the Π̃-matrix is an
estimate of the true
transition probability p_ij such that p̂_ij = n_t(i,j)/N_t(i)
and lim_{N_t(i)→∞} p̂^t_ij = p_ij.
```

(4)
```
for i=1 to |Q|
   for j=1 to |Σ|
      p̂^t_ij = n_t(i,j)/N_t(i)
      Π̃[i][j] ← π̂̃_ij = p̂^t_ij(1 − θ)
   end
end
```

This estimation procedure is conservative since some elements of the $\tilde{\Pi}$-matrix reach the bound before others under the stopping condition of $\forall q_i \in Q, \; \min\{N_t(i)\} \leq N$.

### 7.1 $\tilde{\Pi}$-Matrix Parameter Identification by Simulation

The estimation of event cost $\tilde{\Pi}$-matrix parameters (see Section 7) takes into consideration the number of occurrences of a state $q_i \in Q$ and the number of occurrences of some event $\sigma_j \in \Sigma$ at state $q_i$, of a DFSA ($\mathcal{G}$, $S_1/\mathcal{G}$ and $S_2/\mathcal{G}$) under study. To collect this data, we simulate the production of events. A real-time experiment needs the field data on the distribution of types of faults and also one to decide when to inject a fault. We know the high level events into which $E_{gen}$ maps the lower level signals when a fault is injected and also the capability of the model to respond to these events [20]. Then, assuming that we use the same distribution for injecting the faults in experiments and for producing the corresponding events in simulations, we obtain similar observation sequences. This is the rationale for simulating the production of events in our data collection phase.

#### 7.1.1 Simulation Procedure

We use a uniform random number generator that produces a random number in the range $[0, 1)$ to simulate the production of events. For each state, we designate regions in range $[0, 1)$ which are mapped to events defined on that state. This applies to each DFSA we consider in our simulations. One run of a simulation is defined when the DFSA starts from the initial state and returns back to the initial state, designated as the idle state (see Tables 2 and 3). When a random number is generated, its position in the $[0, 1)$ range decides the event to be produced. The random number produced could lie in a region which maps to an event that is disabled by the control policy. In such a case, the event is not produced, but the random number generation is done again until a random value that maps to an enabled event is generated. This process is repeated at each state. For each event that is produced, the DFSA will make transitions and, thus, we obtain a simulated path an application would take under the influence of software faults. At the end of each such run, we collect the event trajectory (observation sequence or event string) and compute the event production frequency $(n_t(i, j))$ and the state frequency $(N_t(i))$. We give here one such observation sequence: $\sigma_1, \sigma_5, \sigma_2, \sigma_6, \sigma_{10}, \sigma_{10}, \sigma_{10}, \sigma_9$. To account for the occurence of unmodeled events, we consider $\theta = 0.02$. Therefore, each row element of the $\tilde{\mathbf{\Pi}}$-matrix is multiplied by $1 - \theta$, i.e., 0.98.

## 8 Experimental Results and Discussion

As a proof of concept we have implemented a prototype of our fault mitigation model. To test this model, we have used five different computer programs where each program performs simple mathematical operations in a loop. These programs, which are written in the *C* programming language, were injected with overflow errors such as division by zero and memory errors such as segmentation faults. In the following subsections, we report the representative results of controlling these computer programs using the two supervisory control policies $S_1$ and $S_2$, described in Section 3 and the performance results of the DFSAs for plant and the controllers.

### 8.1 Observations of Real-Time Software Control under SCT Framework

The default result of both division by zero and segmentation faults is termination of the process. When the divide by zero exception occurs, the sensor detects the exception. According to both the control policies, the process is not terminated on the very first instance of the exception, implying the event $\sigma_3$ is disabled. To enforce the disabling decision of the supervisor, the sensor inserted its own handlers at runtime by dynamically loading and unloading *kernel modules* and then made the process aware of the exception. The new handlers incremented the instruction pointer so that same exception does not occur again. When the instruction pointer is incremented the first time, it produces a segmentation fault. The sensor intercepted the segmentation fault and then modified the `task_struct` of the process to accommodate new handlers for segmentation faults. Here, too, the handlers

incremented the instruction pointer. Finally, as the application resumed from the exception handling, it continued with its normal execution. When the number of exceptions incurred by the software application was more than that prescribed by the control language, the supervisor permitted the OS to terminate the process.

### 8.2 Performance Measure Results

The event cost matrix is computed using the simulation data (see Section 7.1) for a $\delta = 0.005$, $\varepsilon = 0.075$, and $\theta = 0.02$ by applying the Event Cost Matrix Parameter Estimation procedure (see Section 7 and Appendix B). In this section, we compute the state transition cost matrix $\Pi$ from the event cost $\tilde{\Pi}$-matrix using Definition 7. Each $\Pi$ matrix is followed by the $\overline{\chi}$ vector of each DFSA. We repeat the $\overline{\chi}$ vectors here for the readers ease. The $\overline{\chi}$ matrix of each DFSA contains weights that are assigned to the marked states of the DFSA. For the rationale used in assigning weights to marked states refer to Sections 3.2, 4.1.2, 4.2.2, 6.1, and 6.2. We map the states of the controllers $(S_1/\mathcal{G})$ and $(S_2/\mathcal{G})$ into row indexes of their corresponding $\Pi$ matrices. State $q_1$ of the plant model and $(q_1, x_1)$ of both supervisors are mapped into the first row of their respective $\Pi$-matrices.

The matrix $\Pi_{\mathcal{G}}$ is the $\Pi$-matrix for the uncontrolled plant model $\mathcal{G}$.

$$\Pi_{\mathcal{G}} = \begin{bmatrix} 0.0 & 0.0 & 0.98 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.98 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.2515 & 0.0 & 0.2427 & 0.0 & 0.2466 & 0.2392 \\ 0.0 & 0.1997 & 0.0 & 0.2013 & 0.2055 & 0.0 & 0.3736 \\ 0.0 & 0.1413 & 0.0 & 0.1505 & 0.1542 & 0.0 & 0.5341 \\ 0.0 & 0.1671 & 0.184 & 0.0 & 0.2292 & 0.0 & 0.3997 \\ 0.4887 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.4913 \end{bmatrix}.$$

Our choice of the weights given to the marked states of the DFSA $\mathcal{G}$ are below in the $\overline{\chi}_{\mathcal{G}}$ matrix.

$$\overline{\chi}_{\mathcal{G}} = [0.3 \quad 0.8 \quad 0.8 \quad 0.1 \quad 0.08 \quad 0.4 \quad -0.8]^T.$$

Using the vector space formula $\overline{\mu} = [I - \Pi]^{-1}\overline{\chi}$ (see Section 6.1), we compute the language measure for the DFSA of the uncontrolled plant model and is given below by matrix $\overline{\mu}_{\mathcal{G}}$.

$$\overline{\mu}_{\mathcal{G}} = [8.332 \quad 8.965 \quad 8.195 \quad 7.157 \quad 6.927 \quad 7.564 \quad 6.431]^T.$$

Similarly the $\Pi$, $\overline{\chi}$, and $\overline{\mu}$ matrices for the DFSA $S_1/\mathcal{G}$ are given as follows:

$$\Pi_{S_1/\mathcal{G}} = \begin{bmatrix} 0.0 & 0.0 & 0.98 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.98 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.2591 & 0.0 & 0.2408 & 0.0 & 0.2383 & 0.2417 \\ 0.0 & 0.2454 & 0.0 & 0.0 & 0.2396 & 0.0 & 0.495 \\ 0.0 & 0.3114 & 0.0 & 0.2308 & 0.0 & 0.0 & 0.4378 \\ 0.0 & 0.2495 & 0.2542 & 0.0 & 0.2325 & 0.0 & 0.2438 \\ 0.4959 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.4841 \end{bmatrix}$$

The weights which we assign to the marked states of the DFSA $S_1/\mathcal{G}$ are below in the $\chi_{S_1/\mathcal{G}}$ matrix.

$$\overline{\chi}_{S_1/\mathcal{G}} = [0.3 \quad 0.8 \quad 0.8 \quad 0.1 \quad 0.08 \quad 0.4 \quad -0.8]^T$$

$$\overline{\mu}_{S_1/\mathcal{G}} = [10.178 \quad 10.774 \quad 10.079 \quad 9.00 \quad 9.117 \quad 9.777 \quad 8.232]^T.$$

Similarly the $\Pi$, $\overline{\chi}$, and $\overline{\mu}$ matrices for the DFSA $S_2/\mathcal{G}$ are given as follows:

$$\Pi_{S_2/\mathcal{G}} = \begin{bmatrix} 0.0 & 0.0 & 0.98 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.98 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.2491 & 0.0 & 0.2417 & 0.0 & 0.0 & 0.0 & 0.2475 & 0.2417 \\ 0.0 & 0.2536 & 0.0 & 0.0 & 0.2284 & 0.2618 & 0.0 & 0.0 & 0.2362 \\ 0.0 & 0.2229 & 0.0 & 0.0 & 0.0 & 0.0 & 0.2466 & 0.0 & 0.5105 \\ 0.0 & 0.2569 & 0.0 & 0.2354 & 0.0 & 0.0 & 0.2661 & 0.0 & 0.2216 \\ 0.0 & 0.2092 & 0.0 & 0.0 & 0.2331 & 0.0 & 0.0 & 0.0 & 0.5377 \\ 0.0 & 0.2597 & 0.2491 & 0.0 & 0.0 & 0.2343 & 0.0 & 0.0 & 0.2368 \\ 0.4892 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.4908 \end{bmatrix}$$

The $\overline{\chi}_{S_2/\mathcal{G}}$ matrix given below contains our choice of weights assigned to the marked states of $S_2/\mathcal{G}$.

$$\overline{\chi}_{S_2/\mathcal{G}} = [0.3 \quad 0.8 \quad 0.8 \quad 0.1 \quad 0.1 \quad 0.08 \quad 0.08 \quad 0.4 \quad -0.8]^T$$

$$\overline{\mu}_{S_2/\mathcal{G}} = [10.635 \quad 11.223 \quad 10.547 \quad 9.626 \quad 9.253 \quad 9.605 \quad 9.243 \quad 10.24 \quad 8.65]^T.$$

In each experiment, the initial state is state $q_1$ for $\mathcal{G}$ and state $(q_1, x_1)$ for both $(S_1/\mathcal{G})$ and $S_2/\mathcal{G}$; therefore, we are interested in the first member of each of the $\mu$ matrices. We see that $\mu(L_m(\mathcal{G})) = 8.33$, $\mu(L_m(S_1/\mathcal{G})) = 10.17$, while $\mu(L_m(S_2/\mathcal{G})) = 10.63$. The language measures of the controlled plant model $S/\mathcal{G}$ and $S_2/\mathcal{G}$ show very little change. However, the measure for $S_2/\mathcal{G}$ is large as compared to that of both $\mathcal{G}$ and $S_1/\mathcal{G}$. Therefore, we say that the application under the influence of software faults provides the required service, though degraded better under the supervisor $S_2$ than under $S_1$. Its operational time and the capacity to provide service when affected by faults is also better than without any external control. This is because supervisor $S_2$ disables the events $\sigma_3$ and $\sigma_7$ at the state $(q_5, x_{10})$ in $S_2/\mathcal{G}$, where, otherwise, the application would have been terminated under $S_1$ at state $(q_5, x_8)$. Therefore, the performance of $S_2$ is superior as compared to $S_1$ and $\mathcal{G}$.

## 9 Summary and Conclusions

This paper presents a language-theoretic technique to model and control software systems without any structural modifications to the application and the underlying operating system. A supervisory controller is designed to control the execution of a software application which we use here to mitigate the detrimental effects of faults when the program is in execution. The supervisor directs a software system to a safe state under the occurrence of selected faults or other undesirable events, e.g., low resources and physical memory. We delineate the process of synthesizing a supervisor from a plain English language specification and demonstrate the controllability of each specification. The concept of super-

visory control applied to software systems is implemented under the Red Hat Linux 7.2 Operating System and the experimental results are presented.

We present a procedure for computing the language measure parameters, specifically the event cost $\tilde{\Pi}$-matrix, identified from test data to obtain the performance measures of the supervisors and the uncontrolled plant model. These language measures assure that the performance of the controlled software application is superior to the uncontrolled application. The qualitative analysis using the language measure determines when it is advantageous to run the software application under the control of a given supervisor over another. The control technique augmented with the language measure has a wide applicability to mitigate faults in software systems and to provide a ranking among a family of supervisors.

## Appendix A

## Definitions and Nomenclature

We reproduce definitions and nomenclature of concepts in automata theory from [13], [1] that are generally used in supervisory control theory.

**Definition A.1.** *Language generated—$L(\mathcal{G})$.*
*The language generated by a generator $\mathcal{G}$ in (1) is $L(\mathcal{G}) = \{s \in \Sigma^* | \ \delta(q_0, s) \ is \ defined\}$.*

**Definition A.2.** *Language Marked—$L_m(\mathcal{G})$.*
*The language marked by a generator $\mathcal{G}$ in (1) is $L_m(\mathcal{G}) = \{s \in L(\mathcal{G}) | \ \delta(q_0, s) \in Q_m\}$.*

**Definition A.3.** *Prefix-closure—$pr(L)$.*
*Let $L \subseteq \Sigma^*$ , then $pr(L) = \{s \in \Sigma^* | \ \exists t \in \Sigma^* \ (st \in L)\}$. $L$ is said to be prefix-closed if $L = pr(L)$. In other words, pr(L) contains all the prefixes of the language L.*

**Definition A.4.** *Accessibility.*
*For a DFSA $\mathcal{G}$ given in (1), the set of states reachable from a state $p \in Q$ in $\mathcal{G}$ is denoted by $Re_{\mathcal{G}}(p) = \{q \in Q | \ \exists s \in \Sigma^* \ s.t. \ \delta(p, s) = q\}$. $\mathcal{G}$ is said to be accessible if $Re_{\mathcal{G}}(p) = Q$, i.e., if all the states in $\mathcal{G}$ are reachable from the initial state $q_0$.*

**Definition A.5.** *Coaccessibility.*
*A DFSA $\mathcal{G}$ given by (1) is said to be coaccessible if $\forall q \in Q, Re_G(q) \cap Q_m \neq \emptyset$, i.e., at least one marked state is reachable from each state of $\mathcal{G}$.*

**Definition A.6.** *Trimness.*
*An automaton $\mathcal{G}$ given by (1) is said to be trim if it is both accessible and coaccessible.*

**Definition A.7.** *Synchronous Composition.*
*Synchronous Composition of DFSAs is used to represent the concurrent operation of component systems. Given two DFSAs $M_1 = (Q_1, \Sigma_1, \delta_1, q_{0,1}, Q_{m,1})$ and $M_2 = (Q_2, \Sigma_2, \delta_2, q_{0,2}, Q_{m,2})$, the synchronous composition of $M_1$ and $M_2$ is a DFSA defined as follows: $M = M_1 \| M_2 = (Q, \Sigma, \delta, q_0, Q_m)$, where $Q = Q_1 \times Q_2$; $\Sigma = \Sigma_1 \cup \Sigma_2$; $q_0 = (q_{0,1}, q_{0,2})$; $Q_m = Q_{m,1} \times Q_{m,2}$; $\forall q = (q_1, q_2) \in Q, \sigma \in \Sigma$ the transition function $\delta(\cdot, \cdot)$ is defined as follows:*

$$\delta(q, \sigma) = \begin{cases} (\delta_1(q_1, \sigma), \delta_2(q_2, \sigma)) & \delta_1(q_1, \sigma), \delta_2(q_2, \sigma) \ defined, \sigma \in \Sigma_1 \cap \Sigma_2 \\ (\delta_1(q_1, \sigma), q_2) & \delta_1(q_1, \sigma) \ defined, \sigma \in \Sigma_1 - \Sigma_2 \\ (q_1, \delta_2(q_2, \sigma)) & \delta_2(q_2, \sigma) \ defined, \sigma \in \Sigma_2 - \Sigma_1 \\ undefined & otherwise \end{cases}$$

*and, if $\Sigma_1 = \Sigma_2$, then $L_m(M_1 \| M_2) = L_m(M_1) \cap L_m(M_2)$.*

**Definition A.8.** *Completion—$\overline{M}$.*
*The completion of a DFSA $M = (Y, \Sigma, \alpha, y_0, Y_m)$, given by the DFSA $\overline{M} = (\overline{Y}, \Sigma, \overline{\alpha}, y_0, Y_m)$, where $\overline{Y} = Y \cup \{y_D\}$, with $y_D \notin Y$ ($y_D$ denotes the dump state), and $\forall \overline{y} \in \overline{Y}, \sigma \in \Sigma$*

$$\overline{\alpha}(\overline{y}, \sigma) = \begin{cases} \alpha(\overline{y}, \sigma) & if \ \overline{y} \in Y, \alpha(\overline{y}, \sigma) \ defined \\ y_D & otherwise. \end{cases}$$

**Definition A.9.** *Controllability.*
*For an unsupervised plant model $\mathcal{G}$ given by (1), let $K \subseteq \Sigma^*$ be a set of specifications that restricts the plant's behavior. The language $K$ is said to be controllable with respect to $\mathcal{G}$ and $\Sigma_u$ if $pr(K)\Sigma_u \cap L(\mathcal{G}) \subseteq pr(K)$. This condition on $K$ is called the controllability condition. The controllability condition is equivalent to saying that the supervisor never disables an uncontrollable event in $G$, formally $\forall s \in \Sigma^*, \sigma \in \Sigma_u, \ if \ s \in pr(K), s\sigma \in L(\mathcal{G}), \ then \ s\sigma \in pr(K)$.*

**Definition A.10.** *Supremal Controllable Sublanguage $K^{\uparrow C}$.*
*For an unsupervised plant model $\mathcal{G}$ given by the automaton in (1), let $K \subseteq \Sigma^*$ be a set of specification that restricts the plant's behavior. If the language $K$ is not controllable, then we should find the "largest" sublanguage of $K$ that is controllable, where "largest" is in terms of inclusion. Let $\mathcal{C}_{in}$ be the class of controllable sublanguages ($L'$) of $K$, where*

$$\mathcal{C}_{in} = \{L' \subseteq K | pr(L')\Sigma_u \cap L(\mathcal{G}) \subseteq pr(L')\},$$

*then $K^{\uparrow C} = \bigcup_{L' \in \mathcal{C}_{in}(K)} L'$.*

## Appendix B

## Bound on Experimental Observations

This section presents a stopping rule to determine a bound on the number of experiments to be conducted for identification of the $\tilde{\mathbf{\Pi}}$-matrix parameters. The objective is to achieve a trade off between the number of experimental observations and the estimation accuracy. We propose a stopping rule as presented below.

A bound on the required number of samples is estimated using the Gaussian structure for the binomial distribution that is an approximation of the sum of a large number of independent and identically distributed (i.i.d.) Bernoulli trials. Denoting $\tilde{\pi}_{ij}$ as $p$ and $\hat{\tilde{\pi}}_{ij}$ as $\hat{p}$, we have $\hat{p} \sim \mathcal{N}(p, \frac{p(1-p)}{N})$, where $E[\hat{p}] = p$ and $\mathrm{Var}[\hat{p}] = \sigma^2 \approx \frac{p(1-p)}{N}$, provided that the number of samples $N$ is sufficiently large. Let $\mathbf{X} \equiv \hat{p} - p$, then $\frac{\mathbf{X}}{\sigma} \sim \mathcal{N}(0, 1)$. Given $0 < \varepsilon \ll 1$ and $0 < \delta \ll 1$, the problem is to find a bound $N_b$ on the number of experiments such that $P\{|X| \geq \varepsilon\} \leq \delta$. Equivalently,

$$P\left\{\frac{|\mathbf{X}|}{\sigma} \geq \frac{\varepsilon}{\sigma}\right\} \leq \delta \tag{4}$$

that yields a bound on $N$ as:

$$N_b \geq \left(\frac{\theta^{-1}(\delta)}{\varepsilon}\right)^2 p(1-p), \tag{5}$$

where $\theta(x) \equiv 1 - \sqrt{\frac{2}{\pi}} \int_0^x e^{-\frac{t^2}{2}} dt$. Since the parameter $p$ is unknown, we use the fact that $p(1-p) \leq 0.25 \ \forall p \in [0, 1]$ to obtain a (possibly conservative) estimate of the bound in terms of the specified parameters $\varepsilon$ and $\delta$ as:

$$N_b \geq \left(\frac{\theta^{-1}(\delta)}{2\varepsilon}\right)^2. \quad (6)$$

The above estimate of the bound on the required number of samples, which suffices to satisfy the specified $\varepsilon - \delta$ criterion, is less conservative than that obtained from the Chernoff bound and is significantly less conservative than that obtained from Chebyshev bound [21], which does not require the assumption of any specific distribution of $\mathbf{X}$ except for finiteness of the $r\text{th}$ ($r = 2$) moment.

## ACKNOWLEDGMENTS

The authors thank Vijay Jain for help in running part of the simulations and help in event cost matrix parameter estimation. This work is supported in part by the US Army Research Office under Grant No. DAAD 19-01-1-0646.

## REFERENCES

[1] P. Ramadge and W. Wonham, "Supervisory Control of a Class of Discrete Event Processes," *SIAM J. Control and Optimization,* vol. 25, no. 1, pp. 206-230, 1987.
[2] G. Karsai, A. Ledeczi, J. Sztipanovits, G. Peceli, G. Simon, and T. Kovacshazy, "An Approach to Self Adapive Software Based on Supervisory Control," *Proc. Int'l Workshop Self Adaptive Software,* 2001.
[3] C. Wallace, P. Jensen, and N. Soparkar, "Supervisory Control of Workflow Scheduling," *Proc. Int'l Workshop Advanced Transaction Models and Architectures,* 1996.
[4] Y. Hong, D. Chen, L. Li, and K. Trivedi, "Closed Loop Design for Software Rejuvenation," *SHAMAN—Self-Healing, Adaptive and self-MANaged Systems,* 2002.
[5] Y. Huang, C. Kintala, N. Kolettis, and N. Fulton, "Software Rejuvenation: Analysis, Module and Applications," *Proc. 25th Int'l Symp. Fault-Tolerant Computing,* 1995.
[6] H. Hopcroft, R. Motwani, and J. Ullman, *Introduction to Automata Theory, Languages, and Computation,* second ed. Addison Wesley, 2001.
[7] L. Bauer, J. Ligatti, and D. Walker, "More Enforceable Security Policies," *Proc. Foundations of Computer Security Workshop,* July 2002.
[8] U. Erlingsson and F. Schneider, "SASI Enforcement of Security Policies: A Retrospective," *Proc. New Security Paradigms Workshop,* pp. 87-95, Sept, 1999,
[9] Y. Hong, D. Chen, L. Li, and K. Trivedi, "Enforceable Security Policies," *ACM Trans. Information and System Security,* vol. 3, no. 1, pp. 30-50, 2002.
[10] X. Wang and A. Ray, "A Language Measure for Performance Evaluation of Discrete Event Supervisory Control Systems," *Applied Math. Modelling,* to appear.
[11] A. Ray and S. Phoha, "Signed Real Measure of Regular Languages for Discrete-Event Automata," *Int'l J. Control,* vol. 76, no. 18, pp. 1800-1808, 2003.
[12] A. Surana and A. Ray, "Measure of Regular Languages," *Demonstratio Mathematica,* vol. 37, no. 2, 2004.
[13] C. Cassandras and S. Lafortune, *Introduction to Discrete Event Systems.* 1999.
[14] F. Charbonnier, H. Alla, and R. David, "Supervised Control of Discrete-Event Dynamic Systems," *IEEE Trans. Control Systems Technology,* vol. 7, no. 2, pp. 175-187, 1989.
[15] M. Heymann, "Concurrency and Discrete Event Control," *IEEE Control Systems Magazine,* pp. 103-112, 1990.
[16] D. Bovet and M. Cesati, *Understanding the Linux Kernel.* O'Reilly & Assoc., Jan. 2001.
[17] A. Rubini, *Linux Device Drivers.* O'Reilly & Assoc., June 2001.
[18] W. Stevens, *Unix Network Programming,* vol. 1, second ed. Singapore: Addison-Wesley Longman, 1999.
[19] X. Wang, A. Ray, and A. Khatkhate, "On-Line Identification of Language Measure Parameters for Discrete Event Supervisory Control," *Proc. IEEE Conf. Decision and Control,* pp. 6307-6312, 2003.
[20] V. Phoha, A. Nadgar, A. Ray, J. Fu, and S. Phoha, "Supervisory Control of Software Systems for Fault Mitigation," *Proc. 2003 Am. Control Conf.,* 2003.
[21] M. Pradhan and P. Dagum, "Optimal Monte Carlo Estimation of Belief Network Inference," *Proc. 12th Conf. Uncertainty in Artificial Intelligence,* 1996.

**Vir V. Phoha** received the MS and PhD degrees in computer science from Texas Tech University. He is an associate professor of computer science at Louisiana Tech University in Ruston. His research interests include control of software systems, anomaly detection, Web mining, network and Internet security, intelligent networks, and nonlinear systems. He is a senior member of the IEEE and a member of the ACM.

**Amit Nadgar** received the bachelor's degree in computer engineering from Pune University, India, in 2000 and the master's degree in computer science from Louisiana Tech University in 2003. His research interests are in the application of control theory for fault mitigation in software systems, dynamic goal switching in mobile robots, and in the development of real-time operating systems support for high speed networking applications. He has worked in the area of software design and development for embedded systems creating systems software and firmware.

**Asok Ray** received the PhD degree in mechanical engineering from Northeastern University, Boston, in 1976 and also received graduate degrees in each of the disciplines of electrical engineering, computer science, and mathematics. He joined the Pennsylvania State University in July 1985 and is currently a Distinguished Professor of Mechanical Engineering, a Graduate Faculty of Electrical Engineering, and a Graduate Faculty in the Inter-College Program in Materials. His research experience and interests include control and estimation of dynamical systems in both deterministic and stochastic settings, intelligent instrumentation for real-time distributed processes, health monitoring and fault-accommodating and robust control as applied to aeronautics and astronautics, undersea vehicles and surface ships, and power and processing plants. He is a fellow of the IEEE, a fellow of the ASME, and an associate fellow of the AIAA.

**Shashi Phoha** received the PhD degree from Michigan State University and the MS degree from Cornell University. She is a director of the Information Sciences and Technology Division at the Applied Research Laboratory and a professor of electrical and computer engineering at the Pennsylvania State University. Her research interests are in developing real-time data driven cognitive control-actuation algorithms that harness the capabilities of distributed devices for executing dynamically adaptive dependable missions. Her current work focuses on detection and mitigation of anomalous behavior in nonlinear processes with phase transitions. She is a senior member of the IEEE.