On Self-Stabilizing Systems: An Approach to the Specification and Design of Fault Tolerant Systems *

Stanley D. Young †
Vijay K. Garg
Department of Electrical and Computer Engineering
University of Texas at Austin
Austin, Texas 78712
stanley@pine.ece.utexas.edu

January 6, 1994

Abstract

A self-stabilizing system is one which can recover from transient faults in a finite number of steps. We present a theory for determining if a behavior specification can be satisfied with a self-stabilizing system and if not, then what the largest self-stabilizing subset and smallest self-stabilizing superset of the specification is. The effects of self-stabilizing behavior on the control of discrete event systems is also investigated [1].

Keywords: Discrete Event Systems, Fault Tolerant, Controllability

*Supported in part by NSF CCR-911065, TRW Faculty Assistantship Award.
†Supported in part by a DuPont Fellowship.
[1] A preliminary version of this paper appeared as [16].

1 Introduction

The capability to handle arbitrary faults and startup states is a desirable feature in any real system. A self-stabilizing system is a system which is guaranteed to converge to a legitimate state in a finite number of transitions and thenceforth allow only legal transitions [6, 4, 1]. This capability can be used to overcome transient faults and continue processing while retaining a previously specified legal behavior. The approach taken in this work is to consider the entire behavior of a system and not to restrict our attention so that only the suffixes of the behaviors are legal. Earlier work concentrated of the characterization that the suffix of a behavior would be legal after some finite prefix [6, 4, 1].

The approach taken here is also related to the standard approach to stability in control theory and the stability of discrete event systems. In control theory, time invariant systems are used extensively to describe system behavior. The time invariance provides a type of shift invariance with respect to time [5]; as described in Section 4, self-stabilizing systems are invariant with respect to shifts of the events, which can be related to a shift in the time origin. For discrete event systems, often stability is described in terms of a state trajectory visiting a set of states [3, 12]; the procedures given in Section 4 can use the state machine representation of a system to calculate the self-stabilizing supersets and subsets of a given behavior.

Self-stabilizing systems demonstrate the interaction between computer science, mathematics, and control theory. The computer science community has investigated self-stabilizing systems as a means of building systems which are tolerant of transient faults [6, 4, 1]. Communication protocols are a primary example of such systems which must withstand transient faults. The mathematics community has investigated sofic systems, which are essentially the same as self-stabilizing systems, as the closure under homomorphisms of subshifts of finite type, or Markov chains [13, 15, 8]. Aspects of topological dynamics concerns the study of such systems. The control systems community has investigated the stability of various systems and is interested in designing systems which are inherently stable. The techniques which are used to calculate the self-stabilizing behavior for a system are very useful for the stability analysis and design of discrete event systems [14, 11, 3, 12]. Such techniques determine whether a system is stable or not and can be used in the design of stable systems to provide

the behaviors which must be removed to leave the largest subset of the behavior which is self-stabilizing or the behaviors which must be added to provide the smallest superset of the behavior which is self-stabilizing.

In Section 2, the background material from formal language theory and controllability for discrete event systems is reviewed. In Section 3, the concept of self-stabilization is discussed and the self-stabilizing machine is introduced. In Section 4, shift invariant languages are introduced along with the main results concerning the shift invariant sublanguage and superlanguage of a given language. In Section 5, s-regular systems are introduced and their relation to regular languages and shift invariant systems is given. In Section 6, the effects which shift invariance has on controllability for discrete event systems are investigated. In Section 7, some applications of the results in the paper are given.

We use calculational style of proofs for many of our theorems. A proof that $[A \equiv C]$ will be rendered in our format as

A
$\equiv$ { hint why $[A \equiv B]$ }
B
$\equiv$ { hint why $[B \equiv C]$ }
C.

We also allow implies ($\Rightarrow$) in the leftmost column. For a thorough treatment of this proof format we refer interested readers to [7].

2 Machines, Languages and Control

Languages and machines provide a way of representing the behavior of systems, such as the automaton model of a circuit. We use this modeling technique to describe systems and their specification.

2.1 Machines, Languages, and Shifts

A state machine, M, is represented by a five tuple: $M = (Q, A, \delta, q_0, Q_m)$, where

$$\begin{array}{lcl} Q & = & \text{a set of states,} \\ A & = & \text{a finite set of transition labels or events,} \\ \delta & = & \text{the transition relation, } \delta : Q \times A \rightarrow 2^Q, \\ q_0 & = & \text{the initial state, } q_0 \in Q, \\ Q_m & = & \text{the marked states, } Q_m \subseteq Q. \end{array}$$

The transition relation δ is in general a partial relation; consequently, we use $\delta(q, \alpha)!$ to denote that the transition event α is defined from state q. The marked states signify a subset of the state set which is used to determine acceptance of a given sequence of events. A sequence of events is accepted if the machine executing the sequence stops in a marked state.

A language is a set of sequences of symbols. The symbol set A is associated with the events which can occur in a system. The set of finite sequences which consist of symbols from the set A, along with the empty string, ε, is denoted by A^*. A sequence $s \in A^*$ is called a string. The length of a string $s \in A$ is denoted by $\mid s \mid$. A subset $L \subseteq A^*$ consisting of strings is called a language.

The standard Boolean operations of union, intersection, and complementation can be applied to languages in the standard manner. Some other operations are also useful for manipulating languages. The concatenation of two strings $s, t \in A^*$ is denoted by st, or by $s.t$, with the result being a single string consisting of the symbols of the first string s followed by the symbols of the second string t. For example, if $s = ab$ and $t = cd$, then $st = abcd$. The Kleene closure of a language $L \subseteq A^*$ is denoted by L^* and consists of the set of finite concatenations of strings from $L \cup \{\varepsilon\}$. For example, if $L = \{a\}$, then $L^* = \{\varepsilon, a, aa, aaa, \ldots\}$. Note that the description of A^* given above is consistent with this definition. The reverse of a language L is denoted by L^R and is obtained by reversing each string in the language. For example, if $L = \{ab\}$, then $L^R = \{ba\}$. The (left) shift operator applied to a string shifts each symbol to the left and throws away the symbol in the first or head position and is denoted by $\sigma_l(s)$ where $s \in A^*$. For example, if $s \in A^*$ and $s = abc$, then $\sigma_l(s) = bc$. The right shift operator applied to a string shifts each symbol to the right and throws away the symbol in the last or tail position and is denoted by $\sigma_r(s)$ where

$s \in A^*$. For example, if $s \in A^*$ and $s = abc$, then $\sigma_r(s) = ab$. Usually, we discuss the left shift operator and use $\sigma(s)$ to denote the left shift operator acting on string s.

In the following development, the concept of combining two finite state machines to create a single machine, the product machine, is useful. The product machine is a single machine which can be used to represent the synchronous behavior of two original machines. If $M_1 = \{Q_1, A, \delta_1, q_{1,0}\}$ and $M_2 = \{Q_2, A, \delta_2, q_{2,0}\}$ are two machines with the same event set, A, then the product of the two machines is denoted

$$M_1 \| M_2 = (Z, A, \delta_{\|}, z_0)$$

where,

$$Z = Q_1 \times Q_2 \text{ and } z_0 = (q_{1,0}, q_{2,0}) \in Q_1 \times Q_2,$$

$$\delta_{\|}((q_1, q_2), \sigma)! \Leftrightarrow \delta_1(q_1, \sigma)! \wedge \delta_2(q_2, \sigma)!,$$

$$\delta_{\|}((q_1, q_2), \sigma) = \begin{cases} (\delta_1(q_1, \sigma), \delta_2(q_2, \sigma)) & \text{if defined} \\ \text{undefined} & \text{otherwise.} \end{cases}$$

Note that this is not the most general type of product composition since we require that the event sets be the same; however, this definition is adequate for our purposes. The language generated by the product machine has a specific relation to the languages of the machines from which it is composed. If $M_{\|} = M_1 \| M_2$ then

$$L(M_{\|}) = L(M_1) \cap L(M_2).$$

These relations follow directly from the definition of the transition function of the product machine.

2.2 Prefixes, Suffixes, and Substrings

A string $u \in A^*$ is said to be a prefix of string $s \in A^*$ if there is another string $v \in A^*$ such that $s = uv$. We denote that u is a prefix of s by $u \leq s$. The prefix closure of a language $L \subseteq A^*$, denoted $precl(L)$, is given by

$$precl(L) = \{u \in A^* \mid \exists v \in A^* : uv \in L\}.$$

The prefix closed subset, or sublanguage, of a given language L, denoted by $prered(L)$, as in prefix reduction, is given by

$$prered(L) = \{u \in L \mid \forall v \leq u : v \in L\}.$$

$prered(L)$ and $precl(L)$ are related by

$$prered(L) \subseteq L \subseteq precl(L).$$

Similarly, one can define suffixes, suffix closure, and the suffix closed sublanguage by considering the reverse of the original language. Hence, u is a suffix of string s if $u^R \leq s^R$. The suffix closure of L, denoted by $sufcl(L)$, is defined by

$$sufcl(L) = \{u \in A^* \mid \exists v \in A^* : u^R v^R \in L^R\}.$$

And the suffix closed sublanguage of L, denoted by $sufred(L)$, is given by

$$sufred(L) = \{u \in L \mid \forall v^R \leq u^R : v^R \in L^R\}.$$

$sufred(L)$ and $sufcl(L)$ are related by,

$$sufred(L) \subseteq L \subseteq sufcl(L).$$

The set of substrings of a given string $s \in A^*$ is specified by $sub(s)$ and defined by

$$sub(s) = \{t \in A^* \mid \exists u, v \in A^* : s = utv\}.$$

Lemma 2.1 demonstrates a monotonicity property for $sub()$.

Lemma 2.1

Let $s, t \in A^*$.

$$s \in sub(t) \Rightarrow sub(s) \subseteq sub(t).$$

Proof:
The proof of this is obvious and follows from the definition of $sub(t)$.
Q.E.D.

Lemma 2.2 demonstrates that the $sub()$ function can be constructed by considering the prefix and suffix closures in a recursive manner.

Lemma 2.2

Let $s \in A^*$, and $a \in A$.

1. $sub(sa) = sub(s) \cup sufcl(s).a$,
2. $sub(as) = sub(s) \cup a.precl(s)$.

Proof:
The proof of this lemma is obvious and follows from the fact that all the substrings of sa, or as, consist of the substrings of s and the suffixes followed by a, or a followed by the prefixes of s.
Q.E.D.

A more complete description of machines and languages can be found in [9].

2.3 Controllability for Languages

In many systems, the event set, A, can be partitioned into two sets: A_c and A_u representing controllable and uncontrollable events, respectively. A language K is said to be ***controllable*** with respect to language L if

$$precl(K)A_u \cap L \subseteq precl(K).$$

With the concept of controllable and uncontrollable events, one defines a supervisor for a plant which affects the controllable events which a plant executes based on some specified constraints. A supervisor for a plant P where $L = L(P)$ is a map

$$f : L \to 2^A$$

which specifies a set of enabled inputs which can be applied as a function of the string of previously generated events in L. The closed loop system consisting of a supervisor, f, and plant, P, has the finite closed loop behavior denoted by L_f, and defined as follows:

1. $\varepsilon \in L_f$,
2. $w\sigma \in L_f$ if and only if $w \in L_f, \sigma \in f(w),$ and $w\sigma \in precl(L)$.

Note that the definition of L_f implies that $L_f \subseteq precl(L)$ and that $precl(L_f) = L_f$ [14].

Often, one obtains that a language K is not controllable with respect to language L, and the question arises as to what is the largest sublanguage of K which is controllable with respect to L. As shown in [2], the formula for the supremal controllable sublanguage of K, denoted by $K^{\uparrow}$, with respect to L is given by

$$K^{\uparrow} = K - ((L - K)/A_u)A^*.$$

As indicated by the formula, the controllable sublanguage is constructed from the original desired behavior, K, by subtracting those strings which lead to behaviors which cannot be controlled to remain in K.

A more complete introduction to the control of discrete event systems can be found in [14].

3 Self-Stabilization

A *self-stabilizing machine* (*ssm*) is one which can execute only legal strings regardless of the initial or current state of the system.

Definition 3.1 G is a self-stabilizing state machine if $G = (Q, A, \delta)$ where Q is a set of states, A is a finite set of events, and $\delta : Q \times (A \cup \{\varepsilon\}) \rightarrow 2^Q$.

The transition relation δ can be extended in the standard manner to handle strings, i.e. δ can be extended to $\delta : Q \times A^* \rightarrow 2^Q$ by $\delta(q, s\alpha) = \bigcup_{q' \in \delta(q,s)} \delta(q', \alpha)$ where $s \in A^*$ and $\alpha \in A$.

A string $s \in A^*$ is in the behavior of a *ssm* $G = (Q, A, \delta)$ if and only if there exists a $q \in Q$ such that $\delta(q, s)!$. Hence, the language of a *ssm* G, denoted by $L(G)$, is defined as

$$L(G) = \bigcup_{q \in Q} \{s \in A^* \mid \delta(q, s)!\}.$$

The primary characteristic of a self-stabilizing system is that the strings of events which are generated by the system remain legal, or valid, regardless of the initial state of the system.

Example 3.1 Suppose we need a controller for determining the loading order of cargo onto a pallet subject to the constraint that all packages of

type A precede all packages of type B, which in turn precede all packages of type C. Figure 1 gives the self-stabilizing machine which enforces the packing constraint.

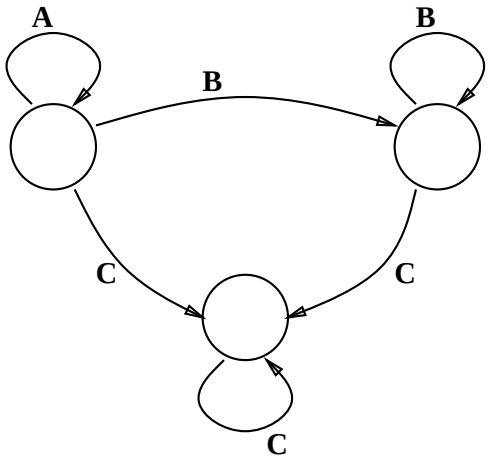

Figure 1: Pallet Packing Controller. (A means load type A, similarly for B and C.)

The system in Example 3.1 is clearly a self-stabilizing machine by inspection of the machine in Figure 1.

Example 3.2 A system which insures that there only an even number of 1's occur between any pair of 0's is given in Figure 2.

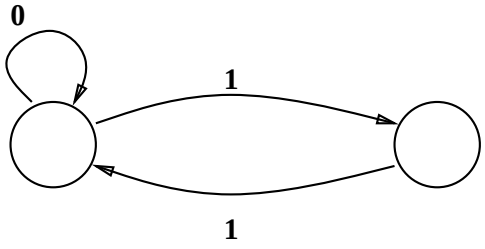

Figure 2: Even System.

Under the interpretation that a new string is started any time the system starts operating, either at startup or after a fault, the system in Example 3.2 is also clearly self-stabilizing. (See Section 4.1 for further discussion of this approach to fault tolerance.)

One technique of describing the desired system behavior is to give a list of behaviors which are not to occur.

Example 3.3 Consider a system which is not to allow either three consecutive zero's or one's. Figure 3 gives a machine which generates such a system.

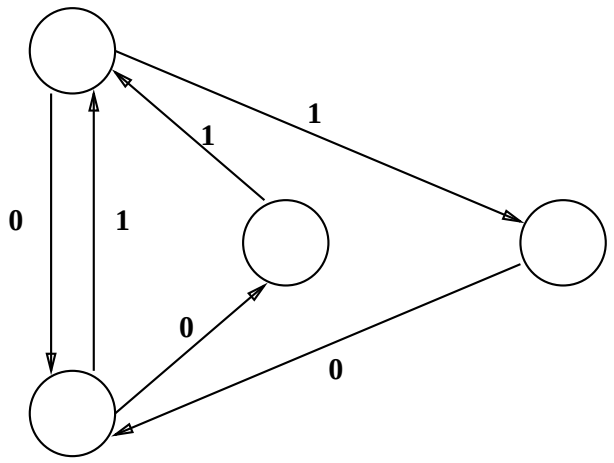

Figure 3: System which disallows three zero's or one's.

4 Shift Invariance

Each application of the shift operator on a string can be viewed as the occurrence of the event which is represented by the symbol at the start, or head of the string. This symbol can also be viewed as the occurrence of the event at the current time. The motivation for studying shift invariant sets is that it would be nice to have a description of the system behavior which does not depend on the definition of the start symbol or time.

4.1 Relation of Shift Invariance to Transient Faults

A transient fault is a fault which changes the state of the system but does not change the structure of the system. There are at least two approaches to specifying a fault tolerant system. In both approaches the system is assumed to be operating normally when a fault occurs.

In one approach, the concatenation of the behavior before and after the fault must remain within the specification. It is easy to show that in the worst case this approach requires the specification to include $(A_K)^*$ where A_K is the set of events which appear in strings in the specification K.

In the second approach, which is the one taken in this paper, "time" is assume to start over after each fault. Hence, the strings for consideration as to whether they meet the specification are those which can occur after a fault, i.e. starting from any state in the system. Note that the concatenation of the strings on both "sides" of a fault are not considered as to whether they satisfy the specification or not, only those occurring after the fault are considered.

4.2 Shift Invariant Theory

Definition 4.1 A language $K \subseteq A^*$ is said to be ***shift invariant*** if

$$\forall a, s : (a \in A) \wedge (s \in A^*) : ((as \in K) \vee (sa \in K)) \Rightarrow (s \in K).$$

The predicate $invariant()$ is used to denote that a set is shift invariant. Hence, $invariant(K)$ denotes that a language K is shift invariant.

Theorem 4.1 demonstrates that shift invariance is preserved under some standard operations such as union, intersection, reversal, concatenation, and Kleene closure. Since shift invariant characteristics are preserved under these operations, large shift invariant systems may be built by combining smaller ones in order to get the desired behavior.

Theorem 4.1

Let $J, K, L \subseteq A^*$.

1. $(\forall i \in \Lambda : invariant(K_i)) \Rightarrow (invariant(\bigcup_i (K_i)))$, where Λ is any nonempty index set.

2. $(\forall i \in \Lambda : invariant(K_i)) \Rightarrow (invariant(\bigcap_i(K_i)))$, where Λ is any nonempty index set.

3. $(invariant(J) \wedge invariant(K)) \Rightarrow invariant(J.K)$.

4. $(invariant(J)) \Rightarrow (invariant(J^R))$.

5. $(invariant(J)) \Rightarrow (invariant(J^*))$.

Proof:
It is clear that the empty string ε is an element of all shift invariant languages. Hence, it is also clear that ε is in the union, intersection, concatenation, reversal, and Kleene closure of those languages. Hence, we will consider only nonempty strings in the following proofs of the different parts of the theorem.

In the following proof, s denotes a string, $s \in A^*$, and a denotes a symbol, $a \in A$.

1. To prove: $(\forall i \in \Lambda : invariant(K_i)) \Rightarrow (invariant(\bigcup_i(K_i)))$, where Λ is any nonempty index set.

$$\begin{array}{ll} & as \in (\bigcup_i K_i) \\ \equiv & \qquad\qquad \{\text{property of union}\} \\ & \exists j : (as \in K_j) \\ \Rightarrow & \qquad\qquad \{\forall i : invariant(K_i)\} \\ & \exists j : (s \in K_j) \\ \equiv & \qquad\qquad \{\text{property of union}\} \\ & s \in \bigcup_i K_i. \end{array}$$

   The proof for $(sa \in \bigcup_i K_i) \Rightarrow (s \in \bigcup_i K_i)$ follows similarly. From these two facts, it follows that $\bigcup_i K_i$ is invariant.

2. To prove: $(\forall i \in \Lambda : invariant(K_i)) \Rightarrow (invariant(\bigcap_i(K_i)))$, where Λ is any nonempty index set.

   This proof is similar to that given for union and is left as an exercise.

3. To prove: $(invariant(J) \wedge invariant(K)) \Rightarrow invariant(J.K)$.

If $as \in J$ or $as \in K$, then, by the invariance of J and K, one has immediately that $s \in JK$. Now, consider the case in which $as \notin J$ and $as \notin K$:

$$\begin{array}{ll} & as \in JK \\ \equiv & \qquad\qquad \{s = uv\} \\ & \exists u, v \in A^* : (au \in J) \wedge (v \in K) \wedge (as = auv) \\ \Rightarrow & \qquad\qquad \{invariant(J)\} \\ & (u \in J) \wedge (v \in K) \\ \equiv & \qquad\qquad \{s = uv\} \\ & s \in JK. \end{array}$$

The proof for $(sa \in JK) \Rightarrow (s \in JK)$ follows similarly. The shift invariance of JK follows from these two facts.

4. To prove: $(invariant(J)) \Rightarrow (invariant(J^R))$.

$$\begin{array}{lll} & as \in J^R & \\ \equiv & & \{\text{definition of } (^R)\} \\ & s^R a \in J & \\ \Rightarrow & & \{invariant(L)\} \\ & s^R \in J & \\ \equiv & & \{\text{definition of } (^R)\} \\ & s \in J^R. & \end{array}$$

The proof for $(sa \in J^R) \Rightarrow (s \in J^R)$ follows similarly. The shift invariance of J^R follows from these two facts.

5. To prove: $(invariant(J)) \Rightarrow (invariant(J^*))$.

This fact follows directly from items (1) and (3) by observing that $J^* = \bigcup_{i=1,\infty} J^i$ where $J^i = \underbrace{J \ldots J}_{i \text{ terms}}$.

Q.E.D.

As Example 4.1 illustrates, shift invariance is not preserved under complementation.

Example 4.1 Let $A = \{a\}$ and $K = \{\varepsilon, a\}$. The complement of K in A^* contains aa but not a; hence, the complement is not shift invariant.

Lemma 4.1 gives the characteristic that, for invariant languages, all the substrings of any string in the language are also in the language.

Lemma 4.1

For language $L \subseteq A^*$:

$$invariant(L) \equiv (\forall s \in A^* : s \in L \Rightarrow sub(s) \subseteq L).$$

Proof:

$\Longleftarrow$:

Assume that L is not invariant.

$$\begin{array}{ll} & \neg invariant(L) \\ \Rightarrow & \qquad\qquad \{\text{definition of } invariant(L)\} \\ & \exists a \in A \wedge s \in A^* : (as \in L \vee sa \in L) \wedge s \notin L \\ \Rightarrow & \qquad\qquad \{\text{definition of } sub(\cdot)\} \\ & sub(as) \not\subseteq L. \end{array}$$

This last implication provides the contrapositive needed to complete the proof.

$\Longrightarrow$:

$$\begin{array}{ll} & invariant(L) \\ \equiv & \qquad\qquad \{\text{definition of } invariant(L)\} \\ & \forall a, s : a \in A \wedge s \in A^* : as \in L \Rightarrow s \in L \\ \equiv & \\ & \forall n : (\forall a, s : a \in A \wedge s \in A^* \wedge \mid s \mid = n : (as \in L \vee sa \in L) \Rightarrow s \in L) \\ \Rightarrow & \qquad\qquad \{\text{induction on } \mid s \mid\} \\ & \forall n : (\forall a, s : a \in A \wedge s \in A^* \wedge \mid s \mid = n : as \in L \Rightarrow sub(as) \subseteq L). \end{array}$$

This last implication follows by a simple induction on the length of s and using Lemma 2.2.

Q.E.D.

By adding enough of the appropriate strings, any set can be made shift invariant. Let the operator $(^{SI})$ denote the operation of adding enough strings to a set to make that set shift invariant.

Definition 4.2 For language $K \subseteq A^*$, we define K^{SI} as the set of strings which are substrings of some string in K, i.e.

$$K^{SI} = \bigcup_{s \in K} sub(s).$$

The relation between K, K^{SI}, and other shift invariant languages containing K is given by Lemma 4.2.

Lemma 4.2

For language $K \subseteq A^*$, K^{SI} is the smallest shift invariant set containing K.

Proof:
We must show that if L is any shift invariant language which contains K, then $K^{SI} \subseteq L$.

$$\begin{array}{ll} & s \in K^{SI} \\ \equiv & \qquad\qquad \{\text{definition of } (^{SI})\} \\ & \exists u, v \in A^* : usv \in K \\ \Rightarrow & \qquad\qquad \{K \subseteq L\} \\ & \exists u, v \in A^* : usv \in L \\ \equiv & \qquad\qquad \{invariant(L)\} \\ & s \in L. \end{array}$$

Q.E.D.
As a result of Lemma 4.2, K^{SI} is called the ***shift invariant superlanguage*** of K.

Theorem 4.2 gives some of the properties of the $(^{SI})$ operator, such as distribution over union, monotonicity, and idempotency.

Theorem 4.2

Let $K \subseteq A^*$.

1. $(\bigcup_i K_i)^{SI} = \bigcup_i (K_i^{SI})$
2. $K_1 \subseteq K_2 \Rightarrow K_1^{SI} \subseteq K_2^{SI}$
3. $(\bigcap_i K_i)^{SI} \subseteq \bigcap_i (K_i^{SI})$
4. $(K^{SI})^{SI} = K^{SI}$

Proof:

1. To prove: $(\bigcup_i K_i)^{SI} = \bigcup_i (K_i^{SI})$.

$$
\begin{array}{ll}
 & s \in (\bigcup_i K_i)^{SI} \\
\equiv & \qquad \{\text{definition of } (^{SI})\} \\
 & \exists u, v \in A^* : (usv \in \bigcup_i K_i) \\
\equiv & \qquad \{\text{property of union}\} \\
 & \exists j : \exists u, v \in A^* : (usv \in K_j) \\
\equiv & \qquad \{\text{definition of } (^{SI})\} \\
 & \exists j : s \in K_j^{SI} \\
\equiv & \qquad \{\text{property of union}\} \\
 & s \in \bigcup_i (K_i^{SI})
\end{array}
$$

2. To prove: $K_1 \subseteq K_2 \Rightarrow K_1^{SI} \subseteq K_2^{SI}$.

$$
\begin{array}{ll}
 & K_1 \subseteq K_2 \\
\equiv & \qquad \{\text{property of union and complement}\} \\
 & K_2 = K_1 \cup (K_2 \setminus K_1) \\
\Rightarrow & \qquad \{(^{SI}) \text{ distribute over union}\} \\
 & K_2^{SI} = K_2^{SI} \cup (K_2 \setminus K_1)^{SI} \\
\equiv & \qquad \{\text{property of union}\} \\
 & K_1^{SI} \subseteq K_2^{SI}.
\end{array}
$$

3. To prove: $(\bigcap_i K_i)^{SI} \subseteq \bigcap_i (K_i^{SI})$.

$$
\begin{array}{ll}
 & \forall i : \bigcap_i K_i \subseteq K_i \\
\Rightarrow & \qquad \{\text{monotonicity of } (^{SI})\} \\
 & \forall i : (\bigcap_i K_i)^{SI} \subseteq (K_i)^{SI} \\
\equiv & \qquad \{\text{property of intersection}\} \\
 & (\bigcap_i K_i)^{SI} \subseteq \bigcap_i (K_i)^{SI}
\end{array}
$$

4. To prove: $(K^{SI})^{SI} = K^{SI}$.

   This fact follows directly from the fact that, for any language L, L^{SI} is the smallest shift invariant language which contains L.

Q.E.D.

Note that the $(^{SI})$ operator does not strictly distribute over intersection as seen in Example 4.2.

Example 4.2 Let languages K_1 and K_2 each have a single string, $K_1 = \{abc\}$ and $K_2 = \{bac\}$. We have that $K_1^{SI} = \{a, b, c, ab, bc, abc\}$ and that $K_2^{SI} = \{b, a, c, ba, ac, bac\}$. Hence, $(\bigcap_i K_i)^{SI} = \emptyset$ and $\bigcap_i (K_i^{SI}) = \{a, b, c\}$.

And, as shown in Example 4.3, $(^{SI})$ does not distribute over concatenation.

Example 4.3 Let languages K_1 and K_2 each have a single string, $K_1 = \{abc\}$ and $K_2 = \{bac\}$. Observe that $bb \in ((K_1)^{SI})((K_2)^{SI})$ but that $bb \notin (K_1 K_2)^{SI}$. Hence, $(^{SI})$ does not distribute over concatenation.

Instead of adding enough strings to a set to make it shift invariant, we can delete strings until the remaining set is shift invariant.

Definition 4.3 For language $K \subseteq A^*$, we define K_{SI} as the set of all strings $s \in K$ such that all substrings of s are also in K, i.e.

$$K_{SI} = \{t \in K \mid sub(t) \subseteq K\}.$$

Lemma 4.3

For language $K \subseteq A^*$, K_{SI} is the largest shift invariant set contained in K.

Proof:
We must show that if L is any shift invariant language which is contained in K, then $L \subseteq K_{SI}$.

$$\begin{array}{lll} & s \in L & \\ \equiv & & \{invariant(L)\} \\ & sub(s) \subseteq L & \\ \Rightarrow & & \{L \subseteq K\} \\ & sub(s) \subseteq K & \\ \equiv & & \{\text{definition of } (_{SI})\} \\ & s \in K_{SI}. & \end{array}$$

Q.E.D.
As a result of Lemma 4.3, K_{SI} is called the ***shift invariant sublanguage*** of K.

Some properties concerning the operator $(_{SI})$, which are similar to those for the $(^{SI})$ operator, are given in Theorem 4.3.

Theorem 4.3

Let $K \subseteq A^*$.

1. $(\bigcap_i K_i)_{SI} = \bigcap_i((K_i)_{SI})$
2. $K_1 \subseteq K_2 \Rightarrow (K_1)_{SI} \subseteq (K_2)_{SI}$
3. $(\bigcup_i K_i)_{SI} \supseteq \bigcup_i((K_i)_{SI})$
4. $(K_{SI})_{SI} = K_{SI}$

The proof of these properties is very similar to the proofs of the properties in Theorem 4.2 and is left as an exercise.

Note that the $(_{SI})$ operator does not strictly distribute over union as seen in Example 4.4.

Example 4.4 Let languages $K_1 = \{abc, \varepsilon\}$ and $K_2 = \{ab, bc, a, b, c, \varepsilon\}$. We have that $(K_1)_{SI} = \{\varepsilon\}$ and that $(K_2)_{SI} = K_2$. Hence, $(\bigcup_i K_i)_{SI} = K_1 \cup K_2$ and $\bigcup_i((K_i)_{SI}) = K_2 \neq (K_1 \cup K_2)$.

Example 4.5 gives the smallest shift invariant language containing and the largest shift invariant language contained in the given language K.

Example 4.5 Let $K = \{abc, ab, a, b, \varepsilon\}$. For this language, we have the following results:

$$K^{SI} = \{abc, ab, bc, a, b, c, \varepsilon\}$$
$$K_{SI} = \{ab, a, b, \varepsilon\}.$$

The following theorem relates the concepts of a self-stabilizing machine, shift invariance, the prefix and suffix closed sublanguages, and the $(^{SI})$ and $(_{SI})$ operators.

Theorem 4.4

Let $K \subseteq A^*$. Then the following are equivalent:

1. K is shift invariant.
2. $K = K^{SI}$.
3. $K = K_{SI}$.

4. $K = prered(K)$ and $K = sufred(K)$.

5. $K = precl(K)$ and $K = sufcl(K)$.

6. There exists a *ssm* G such that $K = L(G)$.

Proof:

To prove: K is shift invariant $\Rightarrow K = K_{SI}$.

Assume $K \neq K_{SI}$.

$$
\begin{array}{ll}
 & K_{SI} \subseteq K \\
\Rightarrow & \qquad\qquad \{K \neq K_{SI} \wedge invariant(K)\} \\
 & \exists s \in K : (s \notin K_{SI} \wedge sub(s) \subseteq K) \\
\Rightarrow & \qquad\qquad \{\text{definition of } (_{SI})\} \\
 & s \in K_{SI}.
\end{array}
$$

This last implication is the desired contradiction which completes the proof of this part.

To prove: K is shift invariant $\Rightarrow K = K^{SI}$.

Assume $K \neq K^{SI}$.

$$
\begin{array}{ll}
 & K \subseteq K^{SI} \\
\Rightarrow & \qquad\qquad \{K \neq K^{SI}\} \\
 & \exists s \in K^{SI} : s \notin K \\
\Rightarrow & \qquad\qquad \{\text{definition of } (^{SI})\} \\
 & \exists u, v \in A^* : usv \in K \\
\Rightarrow & \qquad\qquad \{invariant(L)\} \\
 & sub(usv) \subseteq K \\
\Rightarrow & \qquad\qquad \{\text{definition of } sub(\cdot)\} \\
 & s \in K.
\end{array}
$$

This last implication is the desired contradiction which completes the proof of this part.

To prove: $K = K^{SI} \Rightarrow K$ is shift invariant.

This follows directly from the definitions of $(^{SI})$ and shift invariance.

To prove: $K = K_{SI} \Rightarrow K$ is shift invariant.

This follows directly from the definitions of $(_{SI})$ and shift invariance.

To prove: $(K = K_{SI}) \equiv (K = precl(K) \wedge K = sufcl(K))$.

$$
\begin{array}{ll}
 & K = K_{SI} \\
\equiv & \qquad \{\text{definition of } (_{SI})\} \\
 & \forall s \in K : sub(s) \subseteq K \\
\equiv & \qquad \{\text{definition of } sub(\cdot)\} \\
 & \forall s \in K : ((\forall w \leq s : (w \in K)) \wedge (\forall w^R \leq s^R : (w \in K))) \\
\equiv & \qquad \{\text{definition of } precl(\cdot) \text{ and } suf(\cdot)\} \\
 & K = precl(K) \wedge K = sufcl(K).
\end{array}
$$

To prove: $(K = K_{SI}) \equiv (K = prered(K) \wedge K = sufred(K))$.

This part follows from the previous part by observing that:

$$(K = precl(K)) \equiv (K = prered(K)) \text{ and}$$

$$(K = sufcl(K)) \equiv (K = sufred(K)).$$

Both of these equivalences follow directly from the definitions of $precl(\cdot)$, $prered(\cdot)$, $sufcl(\cdot)$, and $sufred(\cdot)$.

To prove: (There exists a ***ssm*** G such that $K = L(G)) \Rightarrow (K$ is shift invariant).

Let $G = \{Q, A, \delta\}$ be a self-stabilizing machine such that $K = L(G)$. For a string $s \in A^*$, we write $s = s_1 s_2 \ldots s_n$, where $\mid s \mid = n$ and $s_i \in A$.

$$
\begin{array}{ll}
 & s = s_1 s_2 \ldots s_n \wedge s \in L(G) \\
\Rightarrow & \qquad \{\text{definition of } L(G)\} \\
 & \exists q, q' \in Q : (q' = \delta(q, s_1) \wedge \delta(q', s_2 \ldots s_n)!) \\
\Rightarrow & \qquad \{\text{definition of } \delta(\cdot,\cdot)\} \\
 & \exists u \in A^* : u = s_2 \ldots s_n) : (s_1 u \in K) \Rightarrow (u \in K).
\end{array}
$$

The proof for $\exists v \in A^* : v = s_1 \ldots s_{n-1} : (v s_n \in K) \Rightarrow (v \in K)$ is similar. The shift invariance of K follows from these two facts.

To prove: (K is shift invariant) $\Rightarrow$ (there exists a ***ssm*** G such that $K = L(G)$).

We construct a self-stabilizing machine $G = \{Q, A, \delta\}$ which satisfies $K = L(G)$.

Define $Q = \bigcup_{s \in K} sub(s)$.

Hence, for each substring, a different state is defined.

Define the symbol set to be the same used to specify K.

For the transition function, define $\delta(q, \sigma) = q'$, where

for $q = (\sigma_1 \ldots \sigma_{n-1})$ and $s \in K$ such that $s = \sigma_1 \ldots \sigma_{n-1}\sigma$, we have $q' = (s)$.

Otherwise, $\delta(q, \sigma)$ is not defined.

From the construction of the state space, which might be infinite, and the transitions, it is clear that $K = L(G)$.

Q.E.D.

Lemma 4.4 combined with Theorem 4.4 provides a procedure for calculating the language K_{SI} for any language K. This lemma shows that the prefix closed subset of a language, which is then further reduced so that the resulting set is suffix closed, is still prefix closed.

Lemma 4.4

If $L \subseteq A^*$, $K = prered(L)$, and $M = sufred(K)$, then $M = prered(M)$.

Proof:

Assume M is not prefix closed.

$$
\begin{array}{ll}
 & M \neq precl(M) \\
\equiv & \qquad \{\text{property of } precl(\cdot)\} \\
 & \exists s \in M : (\exists t \leq s : t \notin M) \\
\Rightarrow & \qquad \{prered(K) = K\} \\
 & t \in K \setminus M \\
\Rightarrow & \qquad \{sufred(M) = M\} \\
 & \exists t' : ((t')^R \leq t^R \wedge t' \notin K) \\
\Rightarrow & \qquad \{prered(K) = K\} \\
 & \exists u, v : (s = ut'v \wedge t'v \notin K) \\
\Rightarrow & \qquad \{M \subseteq K \wedge sufred(M) = M\} \\
 & s \in K \setminus M.
\end{array}
$$

This last implication provides the contradiction needed to prove the result.

The following diagram gives an indication of the relationship of the strings used in the proof of the lemma. Note that $s_1 = t_1 = u_1$, $t_{|t|} = t'_{|t'|}$, and $s_{|s|} = v_{|v|}$.

$$
\underbrace{\overbrace{u_1 \ldots u_{|u|}}^{u}\underbrace{t'_1 \ldots t'_{|t'|}}_{t' \notin K}\overbrace{v_1 \ldots v_{|v|}}^{v}}_{t \notin M}
$$
$$
\underbrace{\qquad\qquad\qquad\qquad\qquad\qquad}_{s \in M}
$$

Q.E.D.

Procedure 4.1 can be used to calculate the shift invariant sublanguage, K_{SI}, of any language K.

Procedure 4.1

Given language K:

Step 1. Let $L = prered(K)$.

Step 2. Let $M = sufred(L)$.

By Lemma 4.4, $M = prered(M)$, and hence, by Theorem 4.4, $M = K_{SI}$.

Lemma 4.5 combined with Theorem 4.4 provides a procedure for calculating the language K^{SI} for any language K. This lemma shows that the

prefix closure of a language, which is then further increased by the suffix closure operation, is still prefix closed.

Lemma 4.5

If $L \subseteq A^*$, $K = precl(L)$, and $M = sufcl(K)$, then $M = precl(M)$.

The proof of this lemma is similar to that given for Lemma 4.4.

A procedure similar to Procedure 4.1 may be given which takes as input a language K and provides K^{SI} as output.

Procedure 4.2

Given language K:

Step 1. Let $L = precl(K)$.

Step 2. Let $M = sufcl(L)$.

The correctness of this procedure follows from Lemma 4.5 and Theorem 4.4.

5 S-Regular Systems

With the goal of developing a general theory for shift invariant, or self-stabilizing systems, we parallel the general development of event based systems by considering the simplest finite state systems first. More complex systems will be considered in future work. A self-stabilizing machine G with a finite state space Q is denoted as a ***sfsm***. Such systems are referred to as ***sofic systems*** in the literature of topological dynamics [15, 8]. A finite state space allows calculations concerning the system to be effectively accomplished.

A language K is a regular language if and only if it is accepted by an ordinary finite state machine (ofsm) [9]. We define a related concept for self-stabilizing machines.

Definition 5.1 A language $K \subseteq A^*$ is ***s-regular*** if it is regular and shift invariant.

For regular languages, a standard result is that a language K is regular if and only if it is generated by a finite state machine [9]. We have a similar result relating s-regular languages and self-stabilizing machines.

Theorem 5.1

> A language K is s-regular if and only if it is the behavior of a finite state self-stabilizing machine.

Proof:

Since K is s-regular, K is shift invariant and regular. Since K is regular, there is a finite state machine which generates the strings in K. Since K is shift invariant, any substring of a string in K is also in K. The inclusion of these substrings provides that strings which start from any state in the machine are also in K. This characteristic of the language being the same as the set of strings which start from any state in the machine denotes the language generated by a self-stabilizing machine. Hence, there is a self-stablizing machine which generates K. All of this reasoning also applies in the other direction which completes the proof of this theorem.
Q.E.D.

Since unions, intersections, concatenations, and Kleene closures preserve regularity and shift invariance, these operations also preserve s-regularity (see Theorem 4.1 and [9]).

An advantage to restricting attention to finite state systems is that the procedures used to calculate the shift invariant supersets and subsets can be shown to terminate; consequently, for a language K, algorithms are available for calculating K^{SI} and K_{SI}.

Proposition 5.1

> If K is a regular language, then Procedure 4.1 and Procedure 4.2 terminate for input K.

Proof:

Since K is regular, there is a finite state machine $G = \{Q, A, \delta, q_0, Q_m\}$ such that $K = L(G)$. Given the fsm which generates K, calculating the prefix reduction of K takes $O(\mid Q \mid \times \mid A \mid)$ to remove all the nonfinal states and transitions to them. Since K is regular, the prefix reduction of

K is also regular: let M denote this language. Since M is regular there is a finite state machine $G' = \{Q', A, \delta', q_0', Q_m'\}$ such that $M = L(G')$. Given the fsm which generates M, calculating the suffix reduction of M takes $O(\mid A \mid 2^{\mid Q' \mid})$ to reverse the machine. Since Q and Q' are both finite, Procedure 4.1 terminates.

Since K is regular, there is a finite state machine $G = \{Q, A, \delta, q_0, Q_m\}$ such that $K = L(G)$. Given the fsm which generates K, calculating the prefix closure of K takes $O(\mid Q \mid)$ to make all states final states and calculating the suffix closure also takes $O(\mid Q \mid)$ to add an ε transition from the initial state q_0 to all states in order to include all suffixes in the language generated by the machine.
Q.E.D.

An example application of Proposition 5.1 is given in Section 7.

Example 5.1 $L = a^*b^*c^*$ is an example of an s-regular system. The sfsm which generates this behavior is given in Figure 4.

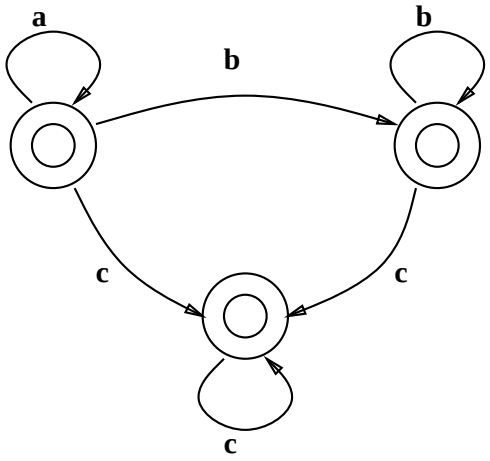

Figure 4: Example s-regular system.

6 Self-Stabilization and Controllability (or Control Without Reset)

The standard discussion of control and languages starts with two languages, K and L, which describe the desired and plant behavior, respectively, and an indication of the controllable and uncontrollable events. As indicated in Section 2.3, the common question concerns the controllability of K with respect to L or the supremal controllable sublanguage of K with respect to L.

The questions relevant to control and languages addressed in this paper concern how the transient faults and a potential lack of a global reset capability in the system can affect controllability. Specifically, if there is no reset capability for the plant, then the plant behavior of interest is actually L^{SI} which might be larger than L. Similarly, the closed loop behavior L_f is actually represented by $(L_f)^{SI}$ which is a superset of L_f.

We make the assumption that a supervisor has access to the state identity of the plant, i.e. a supervisor can observe the current state of the system in order to determine the correct control action.

Example 6.1 and Example 6.2 demonstrates some of the issues which arise in connection with the control of shift invariant systems.

Example 6.1 Let $K = (abc)^*$ and $L = (a(b+d)c+buc)^*$, where $A_u = \{u\}$.

Using the formulas given in Section 2.3, the supremal controllable sublanguage can be calculated to be

$$K^{\uparrow} = (abc)^*.$$

The shift invariant superlanguage of this $K^{\uparrow}$ can be calculated to be

$$(K^{\uparrow})^{SI} = (abc)^*(a + ab + \varepsilon) + (bca)^*(b + bc + \varepsilon) + (cab)^*(c + ca + \varepsilon).$$

Since $K = K^{\uparrow}$, it follows that $K^{SI} = (K^{\uparrow})^{SI}$. Again using the formulas given in Section 2.3, the supremal controllable sublanguage can be calculated to be

$$(K^{SI})^{\uparrow} = (abc)^*(a + ab + \varepsilon) + (cab)^*(c + ca + \varepsilon).$$

Hence, $(K^{\uparrow})^{SI} \neq (K^{SI})^{\uparrow}$.

Example 6.2 Let $K = \{\varepsilon, a, ab, abc, bc, c, b\}$ and $L = K \cup \{bu\}$. In this case, $K_{SI} = K$ and $(K_{SI})^{\uparrow} = \{\varepsilon, a, ab, abc, c\}$. Note that $(K_{SI})^{\uparrow}$ is not shift invariant; consequently, the $(^{\uparrow})$ and $(_{SI})$ operators do not commute.

Considering the observations and examples above, the primary problem considered with respect to control of shift invariant systems can be described as follows:

Problem 6.1 Given a plant and desired behavior, described by languages L and K respectively, find a supervisor f such that $(L_f)^{SI} \subseteq K$.

By Lemma 4.3, Problem 6.1 is equivalent to finding a supervisor f such that $(L_f)^{SI} \subseteq K_{SI}$.

Proposition 6.1 The supervisor f such that $L_f = (K_{SI})^{\uparrow}$ is a solution to Problem 6.1.

Proof:
The proof is clear from the following sequence of set inequalities and the definition of the operators $(^{\uparrow})$ and $(^{SI})$:

$$(L_f)^{SI} = ((K_{SI})^{\uparrow})^{SI} \subseteq (K_{SI})^{SI} = K_{SI} \subseteq K.$$

Q.E.D.

Two related problems concern the problem in which the desired behavior is specified using multiple languages.

Problem 6.2 Given a plant and desired behaviors, described by languages L and K_i respectively, where $i = 1 \ldots n$, find a supervisor f such that $(L_f)^{SI} \subseteq \cup_{i=1\ldots n} K_i$.

Proposition 6.2 The closed loop behavior which results from the union of the separate supervisors f_i created as a result of Proposition 6.1 controlling concurrently and the closed loop behavior which results from the monolithic union of these supervisors are the same, and, additionally, provide solutions to Problem 6.2.

Proof:
The concurrent operation of the supervisors gives the supervisor the behavior $\cup_{i=1\ldots n}(L_{f_i})^{SI}$ and the union provides a supervisor with the behavior

$(\cup_{i=1\ldots n} L_{f_i})^{SI}$. The first inequality below follows from Theorem 4.2. The second follows from Proposition 6.1.

$$(\cup_{i=1\ldots n} L_{f_i})^{SI} = \cup_{i=1\ldots n} (L_{f_i})^{SI} \subseteq \cup_{i=1\ldots n} K_i = K.$$

Q.E.D.

Problem 6.3 Given a plant and desired behaviors, described by languages L and K_i respectively, where $i = 1 \ldots n$, find a supervisor f such that $(L_f)^{SI} \subseteq \cap_{i=1\ldots n} K_i$.

Proposition 6.3 The closed loop behavior which results from the intersection of the separate supervisors f_i created as a result of Proposition 6.1 controlling concurrently and the closed loop behavior which results from the monolithic intersection of these supervisors provide solutions to Problem 6.3.

Proof:
The concurrent operation of the supervisors gives the supervisor the behavior $\cap_{i=1\ldots n} (L_{f_i})^{SI}$ and the intersection provides a supervisor with the behavior $(\cap_{i=1\ldots n} L_{f_i})^{SI}$. The first inequality below follows from Theorem 4.2. The second follows from Proposition 6.1.

$$(\cap_{i=1\ldots n} L_{f_i})^{SI} \subseteq \cap_{i=1\ldots n} (L_{f_i})^{SI} \subseteq \cap_{i=1\ldots n} K_i = K.$$

Q.E.D.

Since the string based controllers are relatively static, as demonstrated by Example 6.1, we might consider those supervisors which are more state based in the sense that they have access to the current state identity in the *ssm* which represents the plant.

A ***state based supervisor*** for a plant P is a map

$$f_Q : Q \rightarrow 2^A$$

which specifies a set of enabled inputs which can be applied as a function of the current state of the plant P.

Let M_L and M_K be the machines which give the plant and desired behavior, respectively. Consider the product of these two machines $M_{||} = M_L \times M_K$. Note that if the machine is considered as a graph, then there

are probably many disconnected components. (Each component may or may not have any defined legal behaviors possible, i.e. the component may consist of a single state; hence, in an actual implementation some reset capability is needed to return the system to a component where legal behavior is possible or some limits are needed on the specification of post-fault states.)

A supervisor for this system must try to inhibit any behaviors which are not allowed in M_K. Towards this end, consider the events which are defined at each state in the plant component of the product machine and are not defined at the desired state component. This set of events must be disabled at that state. For example, let $q = (q_L, q_K)$ be a state in the product machine and q_L be the plant state component and q_K be the desired state component. Let γ_q denote the set of events defined at q_L and not at q_K, i.e.

$$\gamma_q = \{\sigma \in A : \delta_L(q_L, \sigma)! \wedge \neg\delta_K(q_K, \sigma)!\}.$$

Any state for which $\gamma_q \cap A_u \neq \emptyset$ is a bad state. Let G denote the set of bad states in the product machine. The supervisor should control the system away from such states. In order to do this the supervisor should disable all transitions which lead to a bad state or those transitions which lead to states which have uncontrollable transitions defined which lead to a bad state. The region to be avoided can be determined by calculating the ***region of uncontrollable attraction*** for the bad states.

The region of uncontrollable attraction is related to the region of weak attraction, as discussed in [3, 11]. This region is essentially the same as the Z_b region described in [10]. The region of uncontrollable attraction for a specified set of states can be informally described as the set of states from which the system will enter the set of specified states in a finite number of transitions using uncontrolled transitions. The algorithm given below is essentially the same one given in [3] for determining the region of weak attraction modified to calculate the region of uncontrollable attraction.

The ***region of uncontrollable attraction***, $\Upsilon_M(G)$, for a given machine, $M = \{Q, A, \delta, q_0\}$, and a specified subset of states, $G \subseteq Q$, can be determined by the following algorithm [3]:

Algorithm 6.1

Initial Step:

Set $P_0 = G$ and $i = 1$.

Iteration Step:

$$P_i = P_{i-1} \cup \left\{ q \in Q_{\|} : \ \exists \sigma \in A_u : \delta_{\|}(q, \sigma) = q' \in P_{i-1} \ \right\}.$$

Termination Step:

If $P_i = P_{i-1}$,

stop and set $\Upsilon_M(G) = P_{i-1}$,

else

$i = i + 1$ and goto iteration step.

This algorithm builds the region of uncontrollable attraction starting from G. Each iteration of the algorithm adds states to the region defined in the previous iteration. A state is added to the region of uncontrollable attraction only if there is an uncontrollable event σ which describes a transition into the region defined in the previous iteration. The states in $\Upsilon_M(G)$ are well defined, as discussed in [3]. The algorithm is guaranteed to terminate for finite state machines. An efficient algorithm in [11] computes the region of uncontrollable attraction in $O(|Q_{\|}|)$ time.

After calculating the region of uncontrollable attraction, the supervisor must disable any transitions into $\Upsilon_M(G)$. Note that all of these transitions are controllable, otherwise the states would have been included in $\Upsilon_M(G)$.

Proposition 6.4 The state based supervisor, $\hat{f}$, which enforces the rule that transitions into $\Upsilon_M(G)$ are disabled and that for each state q, $\hat{\gamma}_q$ denotes the disabled events, forms the least restrictive state based supervisor for M_L and M_K.

Proof:
Let f denote a state based supervisor for M_L and M_K. We must show that the closed loop behavior allowed by f denoted by M_f is contained in that for $\hat{f}$ denoted by $M_{\hat{f}}$.

This assertion is equivalent to showing that the control patterns generated by f for each state q, γ_q, are such that $\hat{\gamma}_q \subseteq \gamma_q$.

Let $\sigma \in \hat{\gamma}_q$. This definition of σ provides that $\delta(q, \sigma) \in \Upsilon_M(G)$. For f to restrict behavior to K, f must also disable this σ, i.e. $\sigma \in \gamma_q$, in order not to have an uncontrollable sequence which leads out of K.
Q.E.D.

7 Applications

Many different applications may be considered using the results given in this paper. The specific techniques used will depend on how the system is specified and what the goal of the design is.

If the desired behavior of the system is given as the set of "good" finite sequences, then these sequences can be used to construct a finite semigroup and the resultant self-stabilizing machine. As shown in [15, 8], there is an algorithm for converting from a sofic system to a subshift of finite type such that each of the systems has the same entropy; consequently, the information available from the transition matrix of the homomorphic SFT can also be used to describe the sofic system.

7.1 Self-Stabilization for Fault Tolerance

A primary application of the results of this paper is the use of the $(_{SI})$ operator to compute a self-stabilizing machine which satisfies a given general constraint language and is fault tolerant in the sense that only legal behaviors can occur in the computed machine. As an example of this application we consider a specification giving the strings of legal behaviors described by the regular language K. The general procedure is to calculate K_{SI} using Procedure 4.1, generate the sfsm which will generate K_{SI}, and to use standard techniques to implement the finite state machine.

As an example, consider the specification for a power manager for a mobile robot as given in Figure 5. The desired behavior is given by the regular expression

$$K = (f + uf + uuf + uuuf + uuuuf)^*(\varepsilon + u + uu).$$

The objective is to design a self-stabilizing system which will implement this behavior or some subset of it. Following Procedure 4.1, we calculate the prefix closed subset of this language to be the regular expression

$$pred(K) = (f + uf + uuf)^*(\varepsilon + u + uu).$$

Then we calculate the suffix closed subset of this to be the regular expression

$$sufred(pred(K)) = (f + uf + uuf)^*(\varepsilon + u + uu).$$

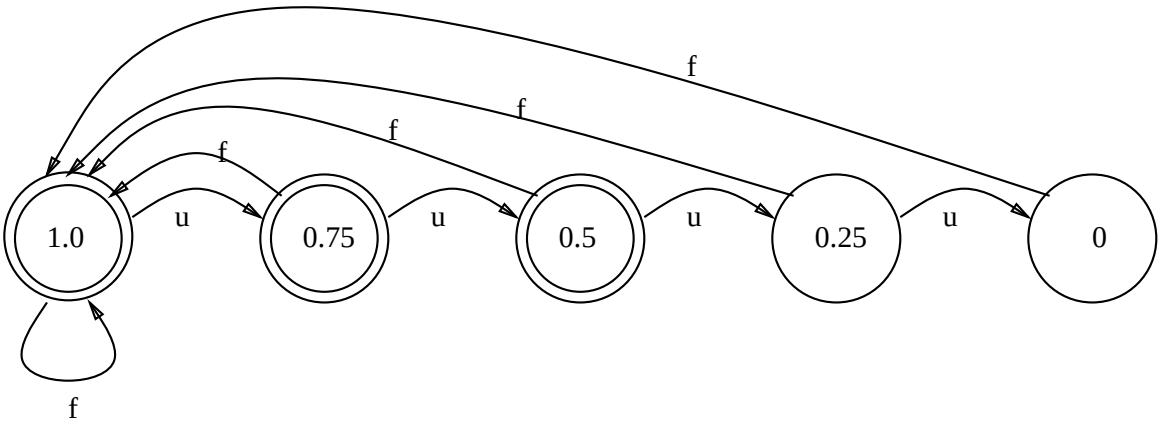

Figure 5: Power controller for robot. (f = fill power reserve, u = use power reserve. Numbers indicate percentage of power reserve left.)

Hence the desired shift invariant sublanguage K_{SI} is given by the regular expression

$$K_{SI} = (f + uf + uuf)^*(\varepsilon + u + uu).$$

7.2 Self-Stabilization and Controllability for Fault Tolerance

In some situations, the system of interest has components which can be controlled and hence might exhibit a more fault tolerant behavior in closed loop operation than in open loop operation.

As an example, consider two flexible manufacturing systems whose input and output streams can be modeled by languages $L_{in,1}$, $L_{out,1}$, $L_{in,2}$, and $L_{out,1}$. (See Figure 6.)

Observe that for the two machines to work correctly, one might specify that $L_{out,1} \cap L_{in,2} \neq \emptyset$. Further if the system is to work without any stops, the extra requirement that $L_{out,1} \subseteq L_{in,2}$ is needed.

In order for FMS_2 to handle faults in FMS_1, any potential output

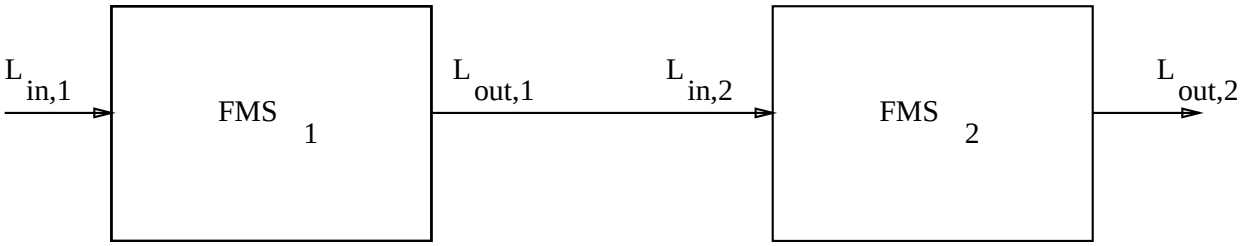

Figure 6: Flexible Manufacturing System.

behavior of FMS_1 must be included in the input behavior of FMS_2, i.e.

$$(L_{out,1})^{SI} \subseteq L_{in,2}.$$

The more restrictive condition that

$$(L_{out,1})^{SI} \subseteq (L_{in,2})_{SI}$$

is required to handle faults in FMS_1 and FMS_2. Hence, for the combined system of the two machines to handle faults, a supervisor, f_1, for the first system must be provided such that

$$((L_{out,1})^{SI})_{f_1} \subseteq (L_{in,2})_{SI},$$

and a supervisor, f_2, for the second system must be provided such that

$$(L_2)_{f_2} = (L_2)_{SI}.$$

If such a supervisor cannot be built, or results in a trivial closed loop behavior, the systems may be examined to determine where the problems are which prohibit the construction of such supervisors.

8 Conclusions

We have addressed the issue of self-stabilizing systems which are those systems which can recover from transient faults.

We have given a technique for determining when a specified system can be implemented as a self-stabilizing system and how to calculate the

largest system which will satisfy the specification and be self-stabilizing. These techniques use standard automata and language theoretic concepts in addition to some concepts from the theory of topological dynamics.

We investigated the effects of shift invariance on the controllability of a desired specification.

Future work consists of applying the techniques in a broader variety of cases, considering how to simplify the description of systems which are synchronously composed, and extensions to languages which consist of infinite strings and languages more general than those described by finite state machines.

It is still an open question concerning the description of a maximal supervisor $\hat{f}$ such that for all supervisors f where $(L_f)^{SI} \subseteq K$ we have that $L_f \subseteq L_{\hat{f}}$.

Also, a primary question for further work concerns how to define the granularity of the events for fault analysis. Example 8.1 demonstrates this issue.

Example 8.1 Consider the behavior specified by $K = ((ab)(cd)^*(ab))^*$. If each symbol a, b, c, and d are considered to be events, then $K_{SI} = \emptyset$; however, if each pair of symbols ab and cd is considered to be an event, then $K_{SI} = (ab)^*$.

It might be necessary to consider more "coarse" events for fault analysis. This reinterpretation of events has the effect of restricting the fault conditions and the states from which and to which the system can go on a fault.

9 Acknowledgements

The authors would like to thank M. Gouda and M. Stafford for beneficial discussions concerning this research.

References

[1] M.S. Abadir and M.G. Gouda. "Self-Stabilizing Digital Circuits". Technical Report TR-90-10, Department of Computer Sciences, University of Texas at Austin, Austin, TX, April 1990.

[2] R.D. Brandt, V.K. Garg, R. Kumar, F. Lin, S.I. Marcus, and W.M. Wonham. "Formulas for calculating supremal controllable and normal sublanguages". *Systems & Control Letters*, 15:111–117, 1990.

[3] Y. Brave and M. Heymann. "On Stabilization of Discrete Event Processes". *Internation Journal Control*, 51:1101–1117, 1990.

[4] G.M. Brown, M.G. Gouda, and C.-L. Wu. "Token Systems that Self-Stabilize". *IEEE Transactions on Computers*, 38(6):845–852, June 1989.

[5] C.T. Chen. *Introduction to Linear System Theory.* Holt, Rinehart, and Winston, New York, NY, 1984.

[6] E.W. Dijkstra. "Self stabilizing systems in spite of distributed control". *Communications ACM*, 17:643–644, November 1974.

[7] E.W. Dijkstra and C.S. Scholten. *Predicate Calculus and Program Semantics.* Springer-Verlag, New York, NY, 1990.

[8] R. Fischer. "Sofic Systems and Graphs". *Monatshefte fur Mathematik*, 80:179–186, 1975.

[9] J.E. Hopcroft and J.D. Ullman. *Introduction to Automata Theory, Languages and Computation.* Addison-Wesley, Reading, MA, 1979.

[10] R. Kumar, V. Garg, and S.I. Marcus. "On Supervisory Control of Sequential Behaviors". *IEEE Transactions on Automatic Control*, 37(12):1978–1985, 1992.

[11] R. Kumar, V.K. Garg, and S.I. Marcus. Language stability and stabilizability of discrete event systems. *SIAM Journal Control Optimization*, 31(5):1294–1320, September 1993.

[12] C.M. Ozveren, A.S. Willsky, and P.J. Antsaklis. "Stability and stabilizability of discrete event dynamic systems". *J. Association for Computing Machinery*, 38(3), July 1991.

[13] W. Parry. "Intrinsic Markov Chains". *Trans. AMS*, 112:55–66, 1964.

[14] P.J.G. Ramadge and W.M. Wonham. "The control of discrete event systems". *Proceedings of IEEE*, 77(1):81–98, January 1989.

[15] B. Weiss. "Subshifts of Finite Type and Sofic Systems". *Monatshefte fur Mathematik*, 77:462–474, 1973.

[16] S.D. Young and V.K. Garg. "On Self-Stabilizing Systems: An Approach to the Specification and Design of Fault Tolerant Systems". In 32^{st} *IEEE Conference on Decision and Control*, San Antonio, TX, 1993.