Shuffle Automata: A Formal Model for Behavior Recognition in Distributed Systems

Peter C. Bates

COINS Technical Report 87-27
January 1987

Laboratory for Distributed Computing
Computer and Information Science Department
University of Massachusetts
Amherst, Massachusetts 01003

This research supported in part by the National Science Foundation under grants MCS-8306327, DCR-8318776, and DCR-8500332. And by the Defense Advanced Research Projects Agency, monitored by the Office of Naval Research under contract NR049-041.

Contents

Shuffle Automata: A Formal Model for Recognizing Behavior in Distributed Systems

ABSTRACT

Event Based Behavior Abstraction is a paradigm for debugging distributed systems which focuses attention on comparing patterns of expected and actual system behavior patterns. Recognizing patterns of behavior in complex distributed systems for the purposes of debugging is a hard task. The *Shuffle Automata* model describes a formalism for recognizing behaviors in distributed systems. The *Simple Shuffle Automata* defines the basic operational characteristics of shuffle automata model. However, simple shuffle automata are not capable of satisfying all the needs of an event based behavior recognizer, so more capable members of the shuffle automata family are defined. A collection of simple shuffle automata formed as a *Basic Shuffle System* and augmented with constraints based on dynamic properties of event models as a *Constrained Shuffle System* will represent all event models described within the event based modelling scheme. This paper develops the shuffle automata family and ties it to event based behavior modelling.

1. Introduction: Shuffle Automata Family

Event Based Behavioral Abstraction (*EBBA*) is a high-level paradigm for debugging distributed systems which focuses on comparing models of actual and expected system behavior. Models of expected behavior are expressed as behavioral patterns in terms of events[1] which represent significant interactions of system components. Recognized instances of these higher level patterns can likewise be considered as events and incorporated as pattern constituents into other behavior models. Thus, the *EBBA* paradigm views behavior recognition as a problem in pattern recognition. For various reasons this approach to behavior recognition is more involved than application of parsing techniques (as in [Fu82]) applied to an input stream of symbols. This paper will present a formalism, the *Shuffle Automata* model, that is useful for describing the meaning of event based behavior abstraction and aids the recognition of behavior models for debugging distributed programs.

Shuffle Automata are a Finite State Automata-like (*FSA*) formalism that comprise a family of machines with a common basic operation. Each higher family member enhances the operational characteristics of the basic model to take in more of the needs posed for behavior recognition. The connection of shuffle automata to behavior recognition is made by considering the symbols that form the input alphabet for the shuffle automata to be the events that make up behavior models.

A shuffle automaton consists of a set of states and a finite state control that effects transitions from an initial state to some final state. Transitions are made from state to state based on the availability of appropriate sequences of input symbols. An important difference between the shuffle automata and *FSA* models is that in order to make transitions in the shuffle automata, the finite state control examines *sets* of input symbols, rather than individual symbols from the input alphabet of the machine. This is a simple means for describing concurrency and collections of behaviors which are only partially ordered in time. In addition, shuffle automata are also capable of describing the usual sequential behaviors: sequence, iteration, and selection.

Another important difference is the capability of shuffle automata to use dynamic properties of events to narrow the focus of a behavioral model. In the *EBBA* paradigm events are tuples consisting of a class that the event is representative of, and a list of attributes that distinguish a particular instance of that class. Shuffle automata can attach functions parameterized by event

[1] *Primitive* events are fundamental system activities; *High-level* events are those expressed in terms of primitive *or* other high-level events. See [Bates83] or [Bates86].

attributes to individual transitions and thus constrain model recognition.

The most basic shuffle automata, the *Simple Shuffle Automata* (*SSA*), bears strong resemblence to the familiar *FSA* generally associated with type 3, or regular, languages [Hopcroft69]. The basic characteristics of user defined behavior models and their recognition by shuffle automata are established by the *SSA*. However, the *SSA* has some inherent limitations that bound its use as a recognizer for behavior models. The *SSA* can become unwieldy for complex expressions, does not fully support hierarchical behavior descriptions, and does not account for the use of event attributes to perform fine filtering of event information.

The first two of these problems are overcome by the *Basic Shuffle System (BSS)*, a more general shuffle automaton, which consists of collections of simple shuffle automata. The *BSS* extension permits high-level events to be easily incorporated into behavior recognition models and simplifies their description. To employ event attributes for filtering, the *Constrained Shuffle System (CSS)*, extends the input alphabet symbol representation to be an event tuple comprised of an event symbol together with a list of attributes. The algorithm that implements the finite state control for a *CSS* includes the use of constraining expressions to effect filtering based on the attributes possessed by an event instance.

The next sections detail the simple shuffle automata model and the relation of the basic model to recognizing behaviors in systems. With this grounding, the more general basic shuffle system is introduced. Several small examples are examined that illustrate the workings as a pattern recognizer. In next to last section, the most general shuffle automata-based recognizer for behavioral patterns, the constrained shuffle system is described.

2. Simple Shuffle Automata

Simple shuffle automata (*SSA*) are similar in form and operation to familiar models for finite state automata. The important difference is that transitions from state to state are based on the simultaneous availability of all elements of a subset of symbols from the machine alphabet. In order to use sets of symbols as transition symbols, the *SSA* finite state control (figure 1) includes an *input register* to hold symbols which have been generated by the symbol sources but not yet used in a transition by the finite state control. This mechanism is the basis for modelling concurrent behavior. Also, the symbol generation sources are considered to operate concurrently and may fill

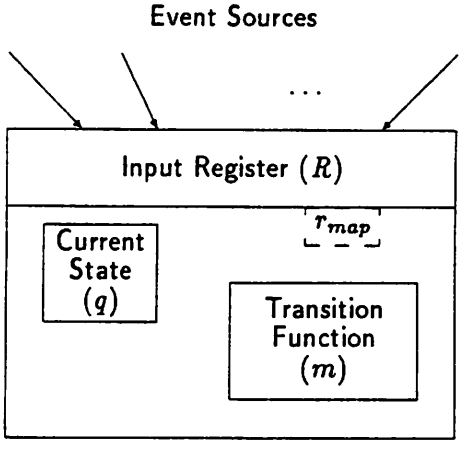


Figure 1: *SSA* control

the input register in parallel. This characteristic is important for distributed programs in which true concurrent program execution is possible in the network.

A simple shuffle automaton is described by the 7-tuple:

$$SSA = (Q, \Sigma, T, m, q_0, F, r_{map})$$

where

Q – finite, nonempty set of states,
Σ – finite alphabet of input symbols,
T – transition sets, $\{t_i \mid t_i \subseteq \Sigma\}, \forall i, j : t_i \neq t_j$,
m – transition function $Q \times T \rightarrow Q$,
q_0 – initial state of the *SSA*, $q_0 \in Q$,
F – set of final states, $F \subset Q$,
r_{map} – input register map, $\Sigma^* \rightarrow \Sigma^*$.

The set of states, Q, the input alphabet, Σ, the transition function, m, and the set of final states, F, are all similar to the like-named elements in any of the various finite automata definitions. The *SSA* control started in state q_0 reads sets of symbols and moves from state to state according to the transition function until it enters a final state $q \in F$. The transition sets, T, are a collection of multisets of input symbols. Each $t_i \in T$ is a multiset whose elements are members of Σ. It is possible that the transition sets overlap, that is, $\forall i, j : t_i \in T, t_j \in T, t_i \cap t_j \neq \{\}$. It is necessary that every transition set element is a member of the alphabet of the simple shuffle automaton, that

is, $\bigcup_{i=1}^{|T|} t_i \subseteq \Sigma$. The transition function m is defined over the transition sets, $t_1, t_2, \ldots, t_m$.

The finite state control associated with a *SSA* is in some state, q, from Q and maintains an input register which contains symbols from Σ. The input register, R, is a multiset which holds the input symbols that have been presented to the shuffle automaton by the symbol generating sources, but not yet consumed by a transition. In a single *move*, the *SSA* in state q examines its input register for a set of symbols which will match any one of the transition sets t_i. If $m(q, t_i)$ is defined then the finite state control enters the state given by $m(q, t_i)$ and applies the r_{map} function to the input register. If the state given by $m(q, t_i)$ is one of the final states F, the *SSA* is deemed to have *recognized* one of the patterns it describes. The recognized pattern corresponds to the list of transition function applications

$$m(q_0, t),\ m(q', t'),\ \ldots, m(q'', t'') \in F.$$

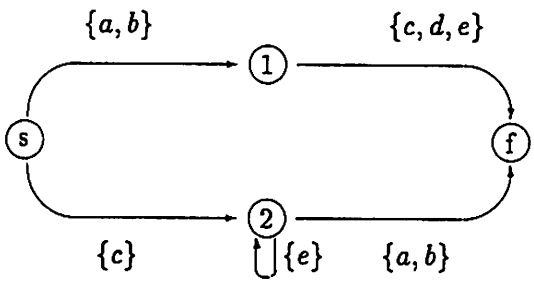
Since shuffle automata are closely tied to the *EBBA* modelling paradigm the set of input symbols which were responsible for the sequence of transitions is called the *event set* for an instance of the pattern described by the *SSA*.

A directed graph may be associated with an *SSA* as follows. Each state of the *SSA* has a corresponding node in the graph. If the *SSA* defines a transition from a state q to state q' on a transition set t_i, then the graph contains a directed arc from node q to node q' labeled with t_i. A set of symbols presented in such a way as to cause the *SSA* to traverse the graph from its initial state to a final state results in recognition of one of the patterns described by the *SSA*. The recognized pattern is the one described by the path taken through the graph. A simple shuffle automaton and its graph are shown in figure 2.

The r_{map} function is used in conjunction with the input register to alter its contents following a transition. Changing the definition of r_{map} leads to different interpretations of the sets of symbols which cause a simple shuffle automaton to make transitions and eventually enter a final state. A simple r_{map} is one which clears the input set following a transition,

$$r_{map} : R \leftarrow \{\}.$$

The effect is to impose a temporal ordering on the symbol subsets which will cause the simple shuffle automaton to enter an accepting state. Symbols not consumed in a transition are discarded and



$$
\begin{aligned}
&Q = \{s, 1, 2, f\} \\
&\Sigma = \{a, b, c, d, e\} \\
&T = \{\{a, b\}, \{e\}, \{c\}, \{c, d, e\}\} \\
&m : \begin{cases} m(s, \{a, b\}) & = 1, \\ m(s, \{c\}) & = 2, \\ m(1, \{c, d, e\}) & = f, \\ m(2, \{e\}) & = 2, \\ m(2, \{a, b\}) & = f \\ otherwise & = \perp \end{cases} \\
&q_0 = s \\
&F = \{f\} \\
&r_{map} : R \leftarrow \{\}
\end{aligned}
$$

Figure 2: A Simple Shuffle Automaton

input symbols	R	*state*	*applicable transition*
–	$\{\}$	s	
$\{a\}$	$\{a\}$	s	
$\{c,e\}$	$\{a,c,e\}$	s	$m(s,\{c\}), R \leftarrow \{\}$
$\{d\}$	$\{d\}$	2	
$\{c\}$	$\{d,c\}$	2	
$\{e\}$	$\{d,c,e\}$	2	$m(2,\{e\}), R \leftarrow \{\}$
$\{b\}$	$\{b\}$	2	
$\{e,e\}$	$\{b,e,e\}$	2	$m(2,\{e\}), R \leftarrow \{\}$
$\{a\}$	$\{a\}$	2	
$\{a,b\}$	$\{a,a,b\}$	2	$m(2,\{a,b\}), R \leftarrow \{\}$
–	$\{\}$	f	

$$event\ set = \{c,e,e,a,b\}$$

Figure 3: *SSA* **operation with** $r_{map} : R \leftarrow \{\}$

not considered for further transitions. The r_{map} that clears the input register at each transition guarantees that a transition from state n to n' which is followed (in time) by a transition from n' to n'' occurs in response to symbol sets that are likewise ordered in time. No symbol that is consumed in the transition from n' to n'' was available for the transition from n to n'. Figure 3 illustrates this for the *SSA* of figure 2.

Another simple r_{map} is one which only removes symbols consumed by a transition $m(q,t_i)$

$$r_{map} :\ R \leftarrow R - t_i.$$

This r_{map} removes the temporal ordering imposed by the previous function. This might be useful when an exact match is not required, but only a pattern that is roughly as specified by the shuffle automaton, or one defective in some way that is yet to be analyzed, is sought.

An important effect of holding unused symbols in the input register is an operational non-determinism. Non-determinism is introduced, not in the definition of a shuffle automaton model, but during its operation. It is always possible to construct a shuffle automaton that appears deterministic in the traditional sense – all transitions from a given state are unique, i.e.

$$\forall\, q,i,j : m(q,t_i) \neq m(q,t_j) \Rightarrow\ t_i \neq t_j.$$

This is the usual restriction for deterministic finite state automata. However, shuffle automata are capable of non-deterministic behavior. It is certainly possible that during operation, a shuffle automaton could be waiting in state q with the above restriction on the transition sets from that state, but both $t_i \subset R$ and $t_j \subset R$. Which transition, m or m', will be taken, depends on other factors.

3. *SSA* as Behavior Recognizers

An *SSA* can be used to recognize behaviors that are specified by event based behavior descriptions. A correspondence between event based behavior descriptions and the *SSA* will be sketched here[2]. A behavior model is specified by an event expression [Shaw79] consisting of event name operands and event operators. Together, the operands and operators specify acceptable sequences of events that fit the model. There are operators to express sequential behavior (indicated with infix o symbols), choice among alternatives (|), concurrent behaviors (Δ), and iteration (postfix *and +).

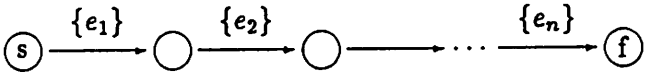
Not all of the needs for event based model recognition are filled by simple shuffle automata. *SSA* are not capable of using constraints to filter events and cannot model complex shuffle expressions without eliminating their concurrency characteristics. The *BSS* described in the next section is capable of describing all event expressions and the *CSS* described in a following section can use constraining expressions to filter the input event stream. A suitable set of tools allows a user to describe behaviors as event based model descriptions which are then recast as recognizing automata and turned loose on the event stream. When a shuffle automaton enters a final state, it has recognized a behavior in the system. In the tool use of abstracted events, the recognized event is sometimes pushed back onto the event stream so it might be incorporated into higher level event models or distributed to remote nodes of a distributed system.

A behavior model specified by an event expression consisting of a series of sequential behaviors such as

$$E = e_1 \circ e_2 \circ \ldots \circ e_n$$

[2] A more complete description containing procedures for transforming models into shuffle automata can be found in [Bates86].

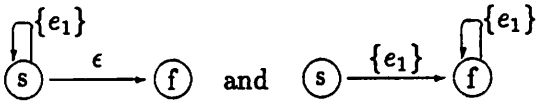
is recognized by a simple shuffle automaton



Likewise event expressions involving the iterative operators such as,

$$E = e_1{}^* \text{ and } E = e_1{}^+$$

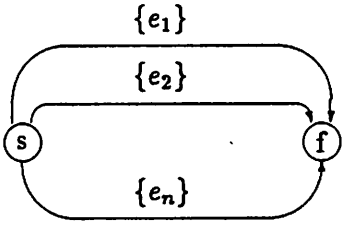
have the respective recognizers



Choice among behaviors, such as in the event expression

$$E = e_1 \mid e_2 \mid \ldots \mid e_n$$

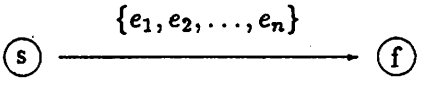
is recognized by a simple shuffle automaton



Finally, the shuffle operator which indicates concurrency for a model

$$E = e_1 \bigtriangleup e_2 \bigtriangleup \ldots \bigtriangleup e_n$$

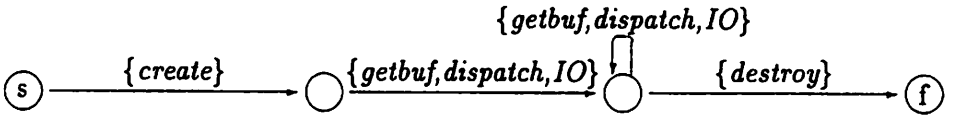
is recognized by the following single transition machine.



As an example, consider a system with five primitive event classes: *create*, *destroy*, *dispatch*, *getbuf*, and *IO*. A behavior model for file movement might be specified

$$fmove = create \circ (getbuf \triangle dispatch \triangle IO)^{+} \circ destroy$$

The recognizer for this model is the simple shuffle automaton



A shuffle automaton is capable of detecting a designated pattern of symbols generated by a set of sources for these symbols. These patterns can be made of symbol sets which are sequential or concurrent in nature. A higher level behavioral model is recognized when its defining shuffle automaton reads a set of events which cause it to enter a final state.

For a simple shuffle automaton

$$S = (Q, \Sigma, T, m, q_0, r_{map})$$

a recognition by S defines an event set

$$E = \{s_i \mid s_i \in \Sigma\}$$

such that there is a series of transitions from the initial state to one of the final states based on the sets of symbols presented to S

$$m(q_0, t_j), m(q^1, t_j^1), \ldots, m(q^m, t_j^m) \in F$$

where

$$\forall i, \exists j, m : s_i \in E \Rightarrow s_i \in t_j^m$$

Recognition of a high-level event by a simple shuffle automaton defines a model of actual system behavior whose constituent events are members of the event set E. For this to work for all event expressions, it is necessary to generalize to the basic shuffle system.

4. Basic Shuffle System

The *SSA* is a good basis for pattern recognition in support of event based behavioral abstraction primarily because of its ability to easily model concurrency and ignore unnecessary information. A *Basic Shuffle System (BSS)* extends the *SSA* to a collection of shuffle automata[3] (*SA*) with a common input alphabet and transition sets that include symbols representing other shuffle automata. A shuffle automaton in the *BSS* operates as does a simple shuffle automaton but is capable of *calling* other shuffle automata in the *BSS* to recognize subparts of a pattern of symbols. For each recognition by a shuffle automaton in the *BSS*, a copy of the symbol that represents the shuffle automaton is created and may be placed into an input register. Thus, shuffle automata sub-machines may be used in the same way as ordinary alphabet symbols to effect transitions from state to state by a shuffle automaton. The call mechanism is similar to that found in the Augmented Transition Network formalism [Woods70], [Bates78]. Among the significant differences is the parallel operation style of the shuffle automata versus the subroutine invocation style of ATN's.

The need for sub-machines also results from an implication of the *SSA* model that the shuffle operator can only connect simple event symbols. The use of a shuffle operator

$$e_1 \triangle e_2 \triangle \ldots \triangle e_n$$

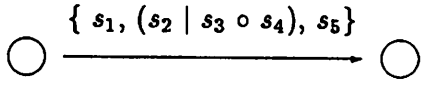
is only possible if all of the e_1, e_2, $\ldots, e_n$ are event symbols. The reason for this is the transition sets have no means to represent objects other than symbols as set members. For example, the event expression

$$s_1 \triangle (s_2 \mid s_3 \circ s_4) \triangle s_5$$

is not directly describable as a simple shuffle automaton because the subexpression

$$\ldots(s_2 \mid s_3 \circ s_4) \ldots$$

needs to be represented in a transition such as



[3]Shuffle automata (*SA*) includes both the *SSA* and *BSS*.

The *BSS* overcomes this difficulty by replacing the $(s_2 \ldots)$ subexpression with a symbol to represent it in the transition set.

The shuffle automata forming the *BSS* are effectively merged into a single pattern recognition entity having a distinguished *root* machine. Once started in the root machine initial state, the *BSS* operates as a set of parallel *SA*. Each *SA* in the *BSS* may be started independently of the others and multiple copies of a *SA* may be active at any time. These possibilities arise because many transition sets marking transitions from a given state may have called for sub-pattern recognition when the state is achieved. From this perspective, the *BSS* is more of a higher organizational level than a machine radically different from the simple shuffle automata.

A *BSS* is defined by the 4-tuple,

$$BSS = (\Sigma, S, T, A)$$

where:

Σ – input alphabet for the *BSS*, $\{s_1, s_2, \ldots, s_n\}$,
S – set of Shuffle Automata, $\{S_i \mid S_i \text{ is a shuffle automaton}\}$,
T – transition sets, $\{t_i \mid t_i \subseteq \Sigma \cup S\}, \forall\, i, j\; t_i \neq t_j$
A – "Active" shuffle automata $A \subseteq S^*$.

The set of Shuffle Automata, S, contains at least the "root" shuffle automaton, S_0. The definition of an individual shuffle automaton, S_i, in the *BSS* is changed slightly from the *SSA* definition to accommodate the input alphabet that is shared by all of the $S_0, S_1, \ldots, S_n$. Each S_i is defined as a 5-tuple,

$$S_i = (Q_{S_i}, m_{S_i}, q_{0S_i}, F_{S_i}, r_{mapS_i})$$

with

Q_{S_i} – set of states for shuffle automaton S_i,
m_{S_i} – transition function, $Q_{S_i} \times T \rightarrow Q_{S_i}$,
q_{0S_i} – starting state for S_i, $q_{0S_i} \in Q_{S_i}$
F_{S_i} – final states for S_i, $F_{S_i} \subset Q_{S_i}$
r_{mapS_i} – input register mapping function for $S_i, (\Sigma \cup S)^* \rightarrow (\Sigma \cup S)^*$

Each component serves the same purpose and has the same meaning as the corresponding components of the *SSA*.

The transition sets, T, tie all of the shuffle automata in the *BSS* together. The definition of transition sets has been extended from the *SSA* definition to include names of other shuffle automata

in the *BSS*. This permits a state to invoke other shuffle automata to supply symbols needed to satisfy a transition set and hence to move from the state. Because of this calling property it is important that the transition sets are not defined in a circular or recursive fashion. To determine if there are cyclic definitions of events create an $n \times n$ connectivity matrix for the shuffle automata in the *BSS*. Label each column with the name of a shuffle automaton, $S_i \in S$; likewise the rows. For each transition set t_j for which $m(q, t_j)$ is defined for S_i, if some shuffle automata symbol $S_k \in t_j$, then indicate in the matrix that S_i is connected to S_k. The reflexive transitive closure of the connectivity matrix indicates whether there are any cycles in the graph.

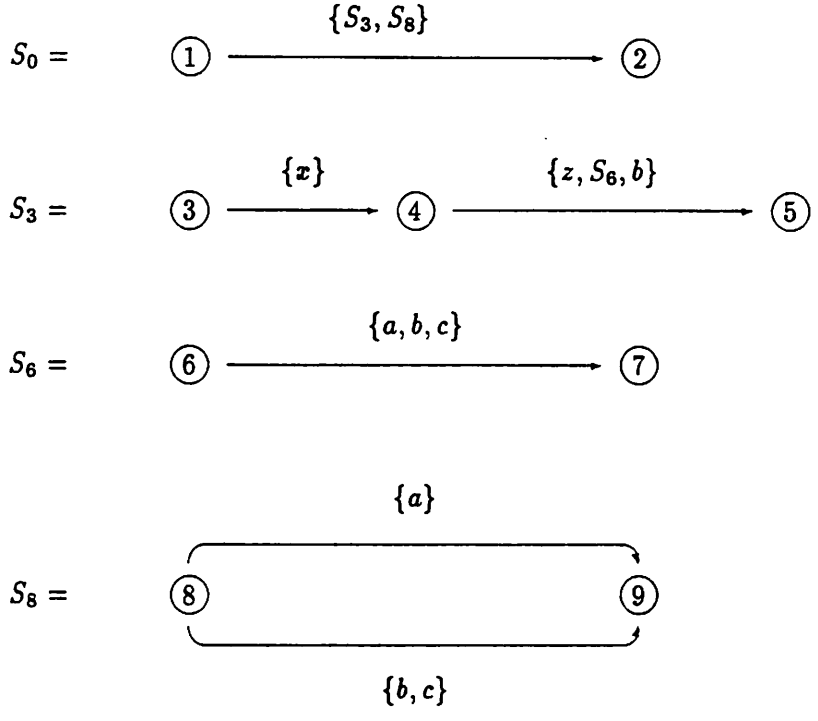
At any time, many *SA* in the *BSS* can be active. To monitor these active shuffle automata, the finite state control contains a set, A, of "active" shuffle automata. This set contains symbols representing those sub-machines which have been called from a state, but have not yet completed their recognition tasks. Figure 4 shows an example basic shuffle system.

A *BSS* begins operation by setting the active shuffle automata set to contain only the root machine S_0 and placing the root machine in its distinguished start state, q_{0S_0}. The root machine proceeds in a similar manner to the *SSA* by comparing the input register to transition sets and performing a transition when a transition set is contained in the input register. However, the finite state control must account for any symbols $S_k \in S$ in the transition sets. When a state q of an active S_k in the *BSS* is entered, each transition $m(q, t_j)$ is examined. Any S_k which are elements of the t_j are called from the state q by adding a copy of the S_k to the active shuffle automata set, placing its finite state control in the starting state q_{0S_k}, and clearing the input register for S_k. Each S_k so started can operate in parallel with other active shuffle automata. Whenever an S_k enters a final state it is removed from the active set and its symbol is added to the input register associated with its caller. The *BSS recognizes* a pattern when the S_0 enters its final state. It is not necessary that the active set be empty for the *BSS* to accept an input.

In the more rigorous explanation of *BSS* operation that follows, S_k^i indicates an instance of shuffle sub-machine S_k with a unique id i. The unique id maps the instance back to the state which made the call. A new identifier can be made from the parent identifier by catenating the current state number for S_k^i (q) to the parent identifier (i)

$$new(i, q) = i \bullet q.$$

The created identifier for a sub-machine can have the state part stripped away and the identifier



$$\Sigma = \{a, b, c, x, z\}$$
$$S = \{S_0, S_3, S_6, S_8\}$$
$$T = \begin{array}{l}\{t_1 = \{S_1, S_8\}, t_2 = \{x\}, t_3 = \{z, S_6, b\},\\ \quad t_4 = \{a, b, c\}, t_5 = \{a\}, t_6 = \{b, c\}\}\end{array}$$
$$A = \{S_0\}$$

$Q_{S_0} = \{1, 2\}$, $q_{0S_0} = 1$, $F_{S_0} = \{2\}$, $M_{S_0} = \{m(1, t_1) = 2\}$.
$Q_{S_3} = \{3, 4, 5\}$, $q_{0S_3} = 3, F_{S_3} = \{5\}$, $M_{S_3} = \{m(3, t_2) = 4, m(4, t_3) = 5\}$.
$Q_{S_6} = \{6, 7\}$, $q_{0S_6} = 6$, $F_{S_6} = \{7\}$, $M_{S_6} = \{m(6, t_4) = 7\}$.
$Q_{S_8} = \{8, 9\}$, $q_{0S_8} = 8$, $F_{S_8} = \{9\}$, $M_{S_8} = \{m(8, t_5) = 9, m(8, t_6) = 9\}$.

Figure 4: Basic shuffle system

of the parent sub-machine returned by a function, *parent(i)*. The more rigorous definition of *BSS* operation follows.

1. the root machine for the *BSS* is preset, and any sub-machines needed to exit the initial state are started,
   $q_{S_0^0} \leftarrow q_0$,
   $R_{S_0^0} \leftarrow \{\}$,
   $A \leftarrow \{S_0^0\}$
   forall $j, l : m(q_0, t_j) \neq \perp$ **and** $S_l \in t_j$
   $A \leftarrow A \cup \{S_l^{new(0,q)}\}$,
   $q_{S_l^{new(0,q)}} \leftarrow q_0 s_l$,
   $R_{S_l^{new(0,q)}} \leftarrow \{\}$
2. As the symbol generator presents symbol sets $\{s_1, s_2, \ldots, s_n\}$, the finite state controls for active shuffle automata evaluate a symbol distribution function to determine which symbols to add to their input registers
   input $\{s_1, s_2, \ldots, s_n\}$,
   forsome $i, j : S^i \in A, s_j \in \{s_1, s_2, \ldots, s_n\}$,
   $R_{S_k^i} \leftarrow R_{S_k^i} \cup \{s_j\}$
3. When the input register of a shuffle automaton in the active set contains one of the outgoing transition sets for the current state of the shuffle automaton, the finite state control for the shuffle automaton makes a transition and alters the input register accordingly,
   if $\exists i, j : m(q_{S_k^i}, t_j) \neq \perp$ **and** $t_j \subseteq R_{S_k^i}$
   then
   $q_{S_k^i} \leftarrow m(q_{S_k^i}, t_j)$,
   $R_{S_k^i} \leftarrow r_{map}(R_{S_k^i})$
   forall $j, l : m(q_{S_k^i}, t_j) \neq \perp$ **and** $S_l \in t_j$
   $A \leftarrow A \cup \{S_l^{new(i,q)}\}$,
   $q_{S_l^{new(i,q)}} \leftarrow q_0 s_i$,
   $R_{S_l^{new(i,q)}} \leftarrow \{\}$
   else goto step 2
4. When a shuffle automata S_k^i in the active set enters a final state, the S_k^i returns to its calling shuffle automaton and is removed from the active set,

if $\exists i : S_k^i \in A$ **and** $q_{S_k^i} \in F$,

then

$A \leftarrow A - \{S_k^i\}$,

if $k = 0$ **then** accept,

$R_{S_k^{parent(i)}} \leftarrow R_{S_k^{parent(i)}} \cup S_k$

5. **goto** step 3

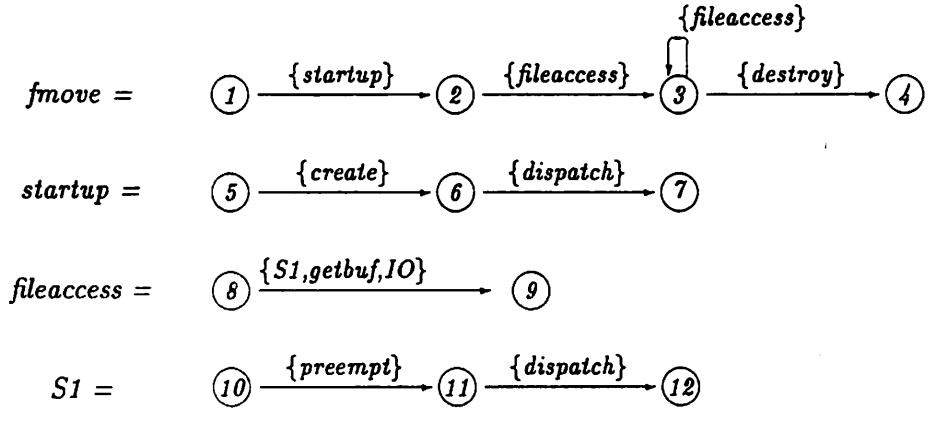
Each $S_i \in S$ in a *BSS* can be shown to be the same as a simple shuffle automata with an appropriately defined alphabet. Viewed in this way, a *BSS* represents a collection of simple shuffle automata, each having its own event set. In the event based modelling perspective some event set elements are primitive events, some are representatives of complex shuffle expression operands, and some are high-level events. The actual behavior model for an event expression represented by a *BSS* is the union of all the event sets bound to the constituent $S_i \in S$ for the basic shuffle system. A recognition by a basic shuffle system occurs if a stream of events presented to the shuffle automaton contains a set of symbols which cause the basic shuffle system root to perform transitions and eventually enter a final state. For example, for the event expression

$$fmove = startup \circ fileaccess^{+} \circ destroy$$

where

$$startup = create \circ dispatch,$$
$$fileaccess = (preempt \circ dispatch) \triangle getbuf \triangle IO.$$

A basic shuffle system to recognize the behavior described by the event expression is:



and has the structure indicated by figure 5.

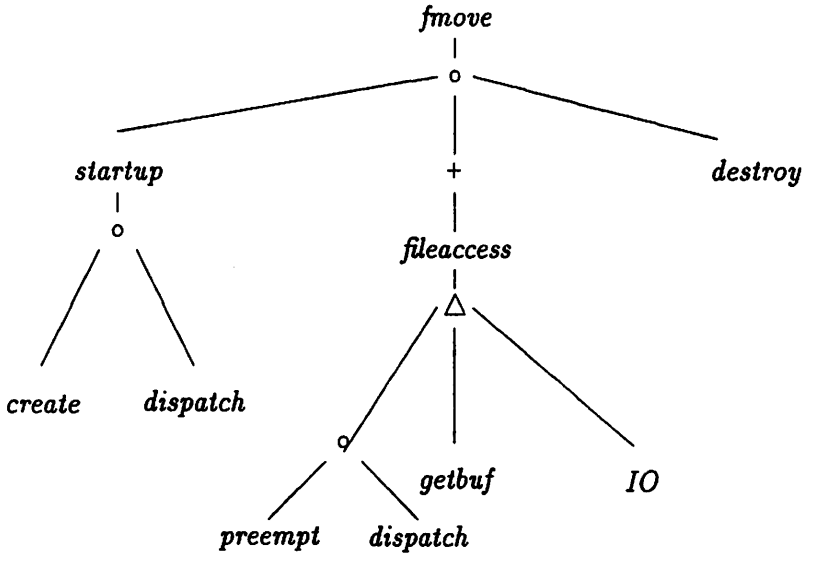


Figure 5: Hierarchical *fmove* model structure

Using the event stream of figure 6, we can trace the activity for the *fmove BSS*. Initially, the *BSS* is placed in its start state (state 1) and a request for *startup* is made. The finite state control for *startup* is placed into its initial state (state 5). The active set for the *fmove BSS* is now

$$\{fmove,\ startup\}.$$

As the event stream enters, *startup* makes transitions and accepts with event set

$$E_{startup} = \{create_1,\ dispatch_1\}.$$

The event symbol for *startup* is placed into the input register for *fmove* allowing *fmove* to take a transition to state 2. From state 2 the *BSS* requests a *fileaccess* event. *Fileaccess* is added to the active list, the finite state control for *fileaccess* is placed into its initial state (state 8), and a request for the subexpression *S1* is made. *S1* is initialized accordingly. The active set is now

$$\{fmove,\ fileaccess,\ S1\}.$$

S1 is recognized with event set

$$E_{S1} = \{preempt_1,\ dispatch_2\}$$

$create_1$
$dispatch_1$
$startup_1$
$preempt_1$
$dispatch_2$
$S1_1$
$preempt_2$
$dispatch_3$
$S1_2$
$getbuf_1$
$getbuf_2$
IO_1
$preempt_3$
$fileaccess_1$
$dispatch_4$
$S1_3$
IO_2
$destroy_1$
$fmove_1$

$E_{startup} = \{create_1, dispatch_1\}$
$E_{S1_1} = \{preempt_1, dispatch_2\}$
$E_{S1_2} = \{preempt_2, dispatch_3\}$
$E_{S1_2} = \{preempt_3, dispatch_4\}$
$E_{fileaccess} = \{S1_1, getbuf_1, IO_1\}$
$E_{fileaccess} = \{S1_3, getbuf_2, IO_2\}$
$E_{fmove} = \{startup_1, fileaccess_1, shutdown_1\}$

Figure 6: Event stream and event sets for *fmove*

and returns an *S1* symbol to the calling state of *fileaccess*. *Fileaccess* takes the transition, *m(8,{S1})* and upon re-entering state 8, requests *S1* again. The *BSS* continues to make transitions, with sub-machines called as needed, until the *fmove* shuffle automaton enters one of its final states and accepts.

The actual behavior model for the *fmove* behavior model is

$$E_{startup} \cup E_{S1_1} \cup E_{S1_3} \cup E_{fileaccess_1} \cup E_{fileaccess_2} \cup E_{fmove}$$

5. Constrained Shuffle System

The extension of the simple shuffle automata to the basic shuffle system model was important because of the need to easily express hierarchical behavior models and complex subexpressions. The pattern recognition model of the *BSS* assumed that events are like symbols in a regular language, featureless and content free. All symbols with the same name are therefore indistinguishable (except for their position in a partially ordered set that defines the event stream). *EBBA* assigns more meaning to events by insisting that events in a class are distinguishable by the attributes they possess. The *Constrained Shuffle System* (*CSS*), introduced here, provides a capability to narrow the focus of a behavioral model by including or excluding events based on the relationships of their attributes.

The first step to providing this capability is to extend the notion of the input symbols to include attributes. A symbol in the alphabet of a CSS is a tuple:

$$(e, a_1, a_2, \ldots, a_n)$$

where e corresponds to the class name of an event and a_1, a_2, ..., a_n correspond to the event's attributes. The event attributes are simple values (the natural numbers) associated with a specific instance of the symbol. All symbols with the same class name possess the same number of attributes. Different instances of a symbol may have the same values bound to their corresponding attribute slots. However, events in a distributed system are all made unique by their combined time and place attributes.

A *CSS* is similar to a *BSS* but contains a set of constraining functions which are defined in terms of the attributes of the input symbols. The finite state control of a *CSS* performs transitions

in a similar manner as a *BSS*, but has the additional task of using the constraining expressions to determine if an input symbol should be included in an event set. A *CSS* then is a 5-tuple

$$CSS = (\Sigma, S, T, C, A)$$

where:

Σ – input alphabet, $\{s_i = (e_i, a_1, a_2, \ldots, a_n) \mid \forall j,\ a_j \in \aleph\}$
S – set of shuffle automata, $\{S_0, S_1, \ldots, S_m\}$,
T – transition sets, $\{t_i \mid t_i \subseteq \Sigma \cup S,\ \forall i, j : t_i \neq t_j\}$
C – set of constraining functions, $\{c_i \rightarrow \{0, 1\}\}$,
A – "Active" shuffle automata $A \subseteq S^*$.

The set of shuffle automata, S, the transition sets, T, and the active set, A, are the same as the corresponding *BSS* elements. The input alphabet Σ is a set of $n+1$-tuples where e_i represents the class name field of the tuple and $a_1, \ldots, a_n$ are the attributes associated with the event.

The constraining functions, C, are defined over the same set of values as the attributes of the event tuples. Without loss of generality, each transition set t_i may have its own constraining function, c_i, associated with it. Each constraining function $c_1, c_2, \ldots, c_n$ in C for the *CSS* is defined in terms of the attributes associated with the event set for the *CSS*. Recall that the event set is a multiset of event instances bound to the event expression member events. In the *CSS*, the symbols $s_1, s_2, \ldots, s_m$ that are bound into the event set, each are formed from a tuple $(e, a_1, a_2, \ldots, a_n)$. For the event set there is a corresponding $m \times n$ array that represents the arguments to the constraining function c_i. The j^{th} row of the array contains the attributes for the j^{th} symbol from event set $E = \{s_1, s_2, \ldots, s_m\}$. The maximum number of columns is determined by the event symbol with the largest number of attributes. It is useful to visualize the event set E the following way

$$E = \left\{ \begin{array}{l} s_1 = (e_1, a_1, \ldots, a_{n_1}) \\ s_2 = (e_2, a_1, \ldots, a_{n_2}) \\ \ldots \\ s_m = (e_m, a_1, \ldots, a_{n_m}) \end{array} \right\}$$

The constraining function for a transition set t_i is defined

$$c_i \begin{pmatrix} a_{1,1} & a_{1,2} & \ldots & a_{1,n} \\ \vdots & & a_{i,j} & \vdots \\ a_{m,1} & a_{m,2} & \ldots & a_{m,n} \end{pmatrix} \rightarrow (0, 1)$$

where each $a_{i,j}$ is the j^{th} attribute of event symbol s_i. For those event set entries with fewer than n attributes, fill their positions in the argument array with 0's. The constraining function c_i is the conjunction of all constraints involving attributes of event symbols defined for t_i.

The constraining functioin c_1 for a transtion set t_i is formed from a conjunction of all the constraint expressions associated with the symbols that are members of t_i. For example, if a set of primitive events have the following templates

(create, node, time, pid, creator)
(dispatch, node, time, pid)
(destroy, node, time, pid, destroyer)
(getbuf, node, time, pid)
(IO, node, time, pid, length, function, device)

an event defined

$$fmove = create \circ (getbuf \triangle dispatch \triangle IO)^{+} \circ destroy$$

with constraints

$$create.node == destroy.node,$$
$$IO.pid == create.pid$$

would define the transition sets

$$t_1 = \{create\},$$
$$t_2 = \{getbuf,\ dispatch,\ IO\},$$
$$t_3 = \{destroy\}$$

with a corresponding set of constraining functions

$$c_i \begin{pmatrix} node_1 & time_1 & pid_1 & creator_1 & 0 & 0 \\ node_2 & time_2 & pid_2 & length_2 & 0 & 0 \\ node_3 & time_3 & pid_3 & 0 & 0 & 0 \\ node_4 & time_4 & pid_4 & length_4 & function_4 & device_4 \\ node_5 & time_5 & pid_5 & destroyer_5 & 0 & 0 \end{pmatrix} \rightarrow \{0,1\}$$

$$c_1 \leftarrow node_1 = node_5 \wedge pid_4 = pid_1,$$
$$c_2 \leftarrow pid_4 = pid_1,$$
$$c_3 \leftarrow node_1 = node_5.$$

The constraining function associated with a transition set is useful because it can effectively eliminate event instances from consideration as event set constituents. This constitutes fine filtering for a behavior model.

Each shuffle automaton S_i of the *CSS* is similar to those of a *BSS* with the exception that the transition function m involves the constraining functions as well as the transition sets. Each $S_i \in S$ for a *CSS* is a 5-tuple

$$S_i = (Q_{S_i}, m_{S_i}, q_{0S_i}, F_{S_i}, r_{mapS_i})$$

where

Q_{S_i} – set of states
m_{S_i} – transition function, $Q_{S_i} \times (T, C) \rightarrow Q_{S_i}$
q_{0S_i} – starting state
F_{S_i} – set of final states
r_{mapS_i} – input register map

Operation of the *CSS* is identical to a *BSS* with added rules to evaluate constraints associated with a transition set. Refer to the operation defined for the *BSS* for a complete description but substitute the following step 3,

3. When the input register of a shuffle automaton in the active set contains one of the outgoing transition sets for the current state of the shuffle automaton, and the transition function for the transition set evaluates to 1, the finite state control for the shuffle automata performs a transition,

if $\exists i, j : m(q_{S_k^i}, t_j) \neq \perp$ **and** $t_j \subseteq R_{S_k^i}$ **and** $c_j = 1$
then
$q_{S_k^i} \leftarrow m(q_{S_k^i}, t_j)$,
$R_{S_k^i} \leftarrow r_{map}(R_{S_k^i})$
forall $j, l : m(q_{S_k^i}, t_j) \neq \perp$ **and** $S_l \in t_j$
$A \leftarrow A \cup \{S_l^{new(i,q)}\}$,
$q_{S_l^{new(i,q)}} \leftarrow q_{0S_i}$,
$R_{S_l^{new(i,q)}} \leftarrow \{\}$
else goto step 4

6. Summary and status

This paper has developed the shuffle automata formalism which is useful for performing pattern recognition in pursuit of behavior behavior modelling for debugging purposes. The shuffle automata model is intended to provide a guideline for recognizing behaviors in complex systems rather than a completely rigorous formal system. A prototype debugging toolset has been constructed to experiment with the *EBBA* paradigm in a real system consisting of networked uni- and parallel

processors. In this toolset, the shuffle automata model forms the core of a distributed model recognizer.

Acknowledgements

The author would like to thank Prof. Jack Wileden of UMASS for his early discussions and criticisms of the shuffle automata model. They were most useful for removing many rough edges and focusing the intent of the model.

REFERENCES

[Aho77] A.V. Aho and J.D. Ullman, *Principles of Compiler Design,* Addison-Wesley, Reading, Massachusetts, 1977.

[Bates78] M. Bates, "The Theory and Practice of Augmented Transition Network Grammars," in *Natural Language Communication with Computers,* ed. L. Bolc, Lecture notes in Computer Science #63, Springer Verlag, 1978.

[Bates83] P. Bates and Jack C. Wileden, "High-Level Debugging of Distributed Systems: The Behavioral Abstraction Approach" *Journal of Software and Systems*, Vol. 3, #4, 1983.

[Bates86] P. Bates, *Debugging Porgrams in a Distributed System Environment*, Ph.D. Dissertation, University of Massachusetts/Amherst, 1986.

[Fu82] K.S. Fu, *Syntactic Pattern Recognition and Applications*, Prentice-Hall, Inc., 1982.

[Hopcroft69] J.E. Hopcroft, J.D. Ullman, *Formal Languages and Their Relation to Automata,* Addison-Wesley, 1969.

[Shaw79] A. Shaw, Software Specification Languages Based on Regular Expressions. Department of Computer Science Report FR-35, University of Washington, 1979.

[Woods70] W.A. Woods, "Transition Network Grammars for Natural Language Analysis," *Communications of the ACM,* Vol. 12, #10, pp. 591-606, Oct. 1970.