A Hybrid State Machine Notation for Component Specification

Alexander Sakharov
mail@sakharov.net
http://alex.sakharov.net

Abstract

A wide range of software units can be classified as state machines. We extend conventional state machine notations by adding regular expressions of events and unions of source states to state machine transitions. Reusable software components are generated from these extended state machine specifications. Component specification and generation are illustrated in Java.

Keywords: finite state machine, regular expression, code generation, component, Java.

1 Introduction

The concept of finite state machines (FSM) is widely exploited in particular applications as well as in commercial development environments. In essence, most real-time systems either are state machines or contain parts that are state machines. Numerous specification notations are based on this concept [Wie]. Programming languages UML [BRJ] and SDL [FO] incorporate FSM notations. Executable code is generated from FSM specifications written in these languages. Use of FSM specifications is growing with growing popularity of UML.

A FSM is defined by the following: a finite set of states, a finite set of event categories and transitions mapping some state-event pairs to other states. Actions are normally associated with transitions. There is considerable similarity between known FSM notations [Wie]. In particular, they all tie transitions to singular events. Since the level of FSM specifications is pretty low, Harel introduced hierarchical statecharts [Ha] in order to reduce the size of the specifications. State machines can be constructed from regular expressions. Regular expressions are a more capacious notation, but in general, they are not a good fit for specification of software units categorized as state machines because regular expressions lack hooks for attaching actions. Still, regular expressions are a proper specification means in cases when actions relate to events.

This article shows how to combine these two kinds of notations. We expand FSM transitions by adding regular expressions of events and by adding unions of source states. Therefore, we associate multiple events with one transition. Use of transitions with regular expressions whose tokens are events may significantly reduce the complexity of FSMs by hiding plenty of states. These transitions with regular expressions of events are turned into internal FSMs that are automatically merged with a host FSM. The use of state unions allows developers to combine multiple transitions in one like composite states in statecharts do [Har].

FSM specifications have been used as source for generation of entire applications. This article addresses generation of generic reusable components from FSM specifications. State machines are components in their nature – they are logical grains to be assembled in applications. State machines process events and have properties and methods. Components generated from FSM specifications can be introspected and customized. They can serve as containers for other components; they provide support for persistence. These components can be used in multi-threaded applications and can be tied with other components both synchronously and asynchronously.

The application pieces modeled by state machines are usually non-visual non-transactional listener components. In contrast to development of client-side visual components [Cow] and server-side transactional components [EJB, GLJ], automation of the development of these non-visual non-transactional listener components has not been much addressed. ActiveX automation servers [Cha] are the only relevant category of components, but those are fairly primitive – they do not handle events.

Generation of code from FSM specifications is illustrated with using Java as the target, specification and implementation language. Generated components are JavaBeans [Eng]. Java introspection facilities make it possible to easily extract information from FSM specifications represented in Java. Otherwise, parsing would be more difficult. Visual development tools like VB [Cow] could be used for specification and generation of FSM components instead of Java. These tools would have to be extended to accept FSM specifications and to generate components from the specifications. We use a TV remote control handler as a running example throughout this paper.

2 Extended Transition Notation

In UML, transitions are specified as arrows connecting two states. UML also allows multiple source and target states in one transition for representing concurrency. A transition string labels each arrow. Its syntax is [OMG]:

<event-signature> [*<guard-condition>*] / *<action-expression>* ^ *<send-clause>*

We are going to use the names of the source and target states as a part of our transition notation instead of labeling arrows. The above transition notation is expanded in two directions.

First, source and target state expressions are introduced. Source state expressions may refer to unions of source states. Each such transition specification denotes multiple transitions - one per every source state. The source state expression is either a state name, or *, or (<*state*>|. . .) or -(<*state*>|. . .). Each of these expressions may be followed by +. The target state expression is either a state name or . (dot). The expression (<*state*>|. . .) denotes the union of the states from the expression. The expression -(<*state*>|. . .) denotes the set of complementary states to ones from the expression. Symbol * is used to denote the set of all named states. Symbol + indicates that respective hidden states should be added to the named states denoted by the state expression. (Hidden states are defined below.) Dot as the target state expression means that the target state is same as the source state.

Second, event expressions are introduced as an extension to event signatures. The event signature is an event name optionally followed by a comma-separated parameter list in parenthesis. The event expression is a regular expression [ASU] whose tokens are event signatures. Event signatures differing in parameters only are not allowed in the same expression. Conventional regular expression syntax is utilized [ASU]. Symbol * denotes zero or more concatenations. Symbol + denotes one or more concatenations. Symbol | denotes union.

The role of event expressions is to simplify FSMs by hiding states. A regular expression can be turned into a deterministic finite automaton [ASU]. Note that deterministic finite automata accepting streams of event names can be viewed as FSMs. This transformation is done for event expressions containing Kleene closure or concatenation [ASU]. The transformation results in internal FSMs (or automata) augmenting their host FSM specified by the developer. The internal FSMs add hidden states to the set of states of the host FSM. These additional hidden states are pairs composed of an explicitly defined state of the host and an internal automaton state. Execution of an internal FSM can be triggered by an event that can be the first token in a string of tokens from the language defined by the regular expression. Once the execution started, the current state is different from any named state. Only one internal FSM can be in a non-start state at any given time because there are no transitions leading from a hidden state of one internal FSM to a hidden state of a different internal FSM.

Any conventional transition defined with using an event signature has a singular action (or procedure) ascribed to it. Several transition actions can be given for a transition defined with an event expression. A separate action can be given for any event occurring in the regular expression. Additionally, notations (<*event-signature*>|. . .): <*action-expression*> and -(<*event-signature*>|. . .):<*action-expression*> can be used to specify a procedure to apply to a set of events. A default action could be provided as well. The default action :<*action-expression*> applies to all events for which no respective procedure is specified. Two more procedures can be specified for a transition with an event expression containing Kleene closure or concatenation: ~premature: <*action-expression*> and ~normal:<*action-expression*>. The premature procedure is executed if the first event making a chain that does not belong to the language defined by the regular expression occurs when the internal FSM was not in an acceptance state [ASU]. The normal procedure is invoked when the respective internal FSM is exited after reaching its acceptance state.

Every transition defined with an event expression is named. Presumably, these names serve as identifiers for providing access to incoming event sequences belonging to the language defined by the regular expression. Here is syntax of the extended transitions:

<name> : *<source-state-expression>* | *<event-expression>* -> *<target-state-expression>* [*<guard-condition>*] /
    <action-expression-list> ^ *<send-clause>*

The action expression list is a comma-separated list in parentheses. Here are sample transition specifications. These are TV control handler transitions in which event enter occurs. The send clauses are omitted from these examples.

```
(-(Off))+ | (power|up|down|mute|one|two|three|four|five|six|seven|eight|nine|zero|enter) -> Off [ errorCode ]
    / (:displayError)

channel : (On|Mute) | ((one|two|three|four|five|six|seven|eight|nine|zero)+ enter) -> .
    / ( (one|two|three|four|five|six|seven|eight|nine|zero):displayChannel, enter:enterChannel,
    ~premature:displayWarning )

incdec : (On|Mute) | (enter (up|down)+) -> .  / (up:incChannel, down:decChannel,
~premature:displayWarning)
```

Note that there could be conflicting transitions defined with using event expressions. They can conflict with the like or with conventional transitions. As usual, only one of them will fire in a single run-to-completion step [OMG]. These conflicts are similar to conflicts in specification of lexical analyzers for generators like lex [ASU]. If more than one regular expression matches an incoming event, the first transition is fired. It is different from lexical analyzers. This early determination is necessary because there could be an action attached to any event, and event processing should not be delayed. Once a transition is selected, the longest lexeme is matched. This is similar to lexical analyzers. Overall, multiple transitions with event expressions whose languages overlap should not normally appear in FSM specifications without proper guards.

3 State Machine Specification in Java

States are specified by their names that are all stored in an array of type String[]. It is assumed that the first element of this array is a start state. Optionally, a stop state is specified. No transition leads to the start state or from the stop state. FSM events are represented in Java as the names of listener interface methods. These methods have one parameter of type EventObject. FSM specifications in Java contain names of listener interfaces and class names of their respective event sources. Transition actions are given by the names of methods implementing them. These methods may have one parameter of type EventObject or may not have parameters. Objects of this type are passed from the respective listener interface methods. The event sources are supposed to implement interface Runnable as well as methods addEventListener and removeEventListener [JBS].

Any FSM component is specified as a class derived from an abstract class called FSM. Class FSM declares variables of type String[] for storing FSM states, event sources and their interfaces. All the aforementioned data members of class FSM are read/write properties. Class FSM defines methods to introspect its properties. These methods exploit the JavaBeans naming design patterns [Eng]. If classes specifying FSMs introduce properties, the developer should define their get/set methods. Presumably, classes specifying FSMs are JavaBeans. They are introspected with using Introspector.getBeanInfo(). Class FSM implements interface Serializable. It is assumed that he classes specifying FSMs do not have non-serializable members.

Here is a FSM specification fragment relevant to the transitions considered earlier:

```
public class TVSpecification extends FSM implements {
Vector channel; Vector incdec;
static { …
  String transitionTmp[][] = {
    { "-(Off)", "(power|up|down|mute|one|two|three|four|five|six|seven|eight|nine|zero|enter)",
        "Off", "[errorCode]", ":displayError" }
    {"name: channel", "(On|Mute)", "(one|two|three|four|five|six|seven|eight|nine|zero)+ enter", ".",
      "(one|two|three|four|five|six|seven|eight|nine|zero): displayChannel", "enter: enterChannel",
      "~premature: displayWarning" },
    {"name: incdec", "(On|Mute)", "enter(up|down)+", ".", "up: incChannel", "down: decChannel",
      "~premature: displayWarning" },
  }; transition = transitionTmp;
… } }
```

4 Application Generation

Application generation is performed via introspection of FSM specification classes. The names of specification classes for all synchronously connected FSMs constitute the set of generator parameters. The generator creates a sub-class for every class specifying a FSM and also builds one event adapter class for the whole set of synchronously connected FSMs. The sub-classes generated from FSM specifications comprise implementations of the listener interfaces specified in their super classes. All generated classes are JavaBeans [Eng]. The generated sub-classes can be customized with respect to developer-defined properties at run time. The generated sub-classes are Serializable.

For the sake of efficiency, events and states are represented by integers in generated code. Implementations of listener interface methods in the sub-classes that are generated from FSM specifications have the form of switch statements whose cases are FSM states including pairs with hidden states. Code for every branch in these switch statements contains calls of respective transition actions. This code also contains fragments generated for relevant transitions with event expressions. Assignments changing the state appear in this code as well. Transition actions for guarded transitions are called within if … else if … else … statements.

```
public class TVSpecificationSub extends TVSpecification implements TVCommunication { …
public void enter( EventObject evt ) { switch (stateIndex) { … } }
… }
```

The generated adapter class is Runnable. It can be asynchronously connected with any other

Runnable classes including other adapter classes. The adapter class fires event sources of its FSMs and registers itself as a listener to the event sources. The adapter implements methods of all listener interfaces specified in the set of relevant FSM components. It calls methods of the same names of the sub-classes generated from FSM specifications. Each of these adapter methods queues incoming events. These event procedures should quickly return in order to avoid loosing events.

```
public void enter(EventObject e) {
    synchronized (eventQueue) { synchronized (indexQueue) {
        eventQueue.addElement(e); indexQueue.addElement(new Integer(<enter>));
} } }
```

The method run of the generated adapter class is implemented as an infinite loop. This loop iterates over the queue of events or waits for events when the queue is empty. The default order of dequeuing events is FIFO, which could be overridden by the developer by means of providing method dequeue in a derivation of the generated adapter class. This method yields the index of the selected event in the queue.

Presumably, method dequeue also implements control flow among synchronously connected components. Hence it should be overridden if the adapter serves multiple FSM components. Method dequeue determines whether a component should be activated when it is inactive and its event occurs. Here are a few examples of component inter-relation that could be implemented in dequeue. Some components may retain control till their completion. Components may or may not resume their execution from a non-start state. Some component may play the role of a container, that is, its sub-components always yield control to their container component upon completion. Method dequeue also may asynchronously start other adapters or foreign Runnable components. Joins with asynchronously called components can be performed in dequeue as well.

In case of multiple components, pairs of methods named begin<component> and end<component> should be overridden in a class derived from the generated adapter. The first of them is called when the respective component is activated. The second method is invoked when this component reaches its stop state. Presumably, these methods implement data interchange among synchronously connected components.

```
for (;;) {
    boolean flag = true; int eventIndex; EventObject eventValue;
    while ( flag ) { synchronized (indexQueue) { flag = indexQueue.isEmpty(); } }
    synchronized (indexQueue) { synchronized (eventQueue) {
        int n = dequeue();
        eventIndex = ((Integer)(indexQueue.elementAt(n))).intValue(); indexQueue.removeElementAt(n);
        eventValue = (EventObject)(eventQueue.elementAt(n)); eventQueue.removeElementAt(n);
    } }
    switch( eventIndex ) { …
        case <enter>: if (component1.isStart()) beginTVSpecification(); component1.enter(eventValue);
            if (component1.isStop()) { endTVSpecification(); component1.reset(); } break;
} }
```

5 Internal FSMs

Transitions based on regular expressions containing Kleene closure or concatenation are turned into internal FSMs that are put together with developer-defined FSMs. Transitions with event expressions containing unions only are unfolded into multiple conventional transitions. Generation of the internal FSMs is done by application of the algorithm converting regular expressions into deterministic finite automata from [ASU]. These generated internal FSMs are minimized after that, i.e. they are transformed into equivalent FSMs with the minimal number of states by using the minimization algorithm from [ASU]. Additionally, if there are transitions in a generated internal FSM leading to its start state, then a new start state is created and added to the internal FSM in order to avoid ambiguity in determining whether the internal FSM is active or its host. The transitions of this newly added start state are obtained from the transitions of the original start state by replacing the source start state with this new start state. With this extra state, an internal FSM is active if and only if the current state is not the start one provided that no transition is in progress.

The conversion algorithm from [ASU] generates internal FSMs with three states for the second and third transitions from the sample specification. The first internal state machine has transitions leading from its start state to the second state and from the second state to itself for all events marked by digits. Event enter occurs in a transition leading from the second state to the third state. The second internal state machine has a transition leading from its start state to its second state for event enter. Events up and down occur in transitions from the second state to the third state and from the third state to itself. These internal FSMs do not undergo further changes. The third states in both machines are acceptance states.

Code fragments implementing operation of internal FSMs are branches in switch statements constituting bodies of listener interface implementations in generated sub-classes of FSM specifications. Each branch corresponds to a pair combining an explicitly defined state and a hidden state. Each transition of an internal FSM augments the vector of events for the internal FSM, calls the respective transition action method, and updates FSM's state. Note that some branches contain code for resetting internal FSMs as well as premature/normal exit procedure calls.

The following example illustrates method enter from the sub-class generated from the specifications under consideration:

```
void enter(EventObject e) {
    switch (stateIndex) {
        case <On>:
            if (errorCode ) { displayError(); stateIndex = <Off>; break; }
            incdec.addElement(new Integer(<enter>)); stateIndex = <On,Incdec:1>; break;
        case <On,Channel:1>:
            if (errorCode ) { displayError(); stateIndex = <Off>; break; }
            channel.addElement(new Integer(<enter>)); enterChannel(); stateIndex = <On,Channel:2>; break;
        case <On,Channel:2>:
            if (errorCode ) { displayError(); stateIndex = <Off>; break; }
            channel.removeAllElements(); incdec.addElement(new Integer(<enter>)); stateIndex =
    <On,Incdec:1>; break;
        case <On,Incdec:1>:
            if (errorCode ) { displayError(); stateIndex = <Off>; break; }
            displayWarning();
            incdec.removeAllElements(); incdec.addElement(new Integer(<enter>)); stateIndex =
    <On,Incdec:1>; break;
        case <On,Incdec:2>:
            if (errorCode ) { displayError(); stateIndex = <Off>; break; }
            incdec.removeAllElements(); incdec.addElement(new Integer(<enter>)); stateIndex =
    <On,Incdec:1>; break;
        case … // other states – code for Mute is similar to above
        default: break;
} }
```

6 Summary

Regular expressions can be mapped to FSMs, but the notation of regular expressions has not been used in specification of FSM transitions. In our hybrid FSM notation, these two specification means – FSMs and regular expressions - are used in concert. Normally, developers have to define a vast number of states and transitions in order to programme a real system. Use of regular expressions of events advances the level of specification by hiding extraneous states. Due to event expressions, the number of developer-specified states can be much less than the total number of states. Use of unions of states also advances the level of specification by reducing the number of transition records.

In spite of the fact that many applications have plenty of non-visual non-transactional listener components that are naturally represented as FSMs, FSMs have never been treated as a component class. We describe how extended FSM specifications can be turned into reusable components that are glued in applications via use of generated adapters. Neither UML nor SDL offer means to generate reusable components from FSM specifications.

Harel introduced statecharts that made it possible to deal with complexity of real systems by structuring FSMs through hierarchies of composite states [Har]. UML [BRJ] adopted Harel's statecharts. Use of multiple synchronously connected FSM components along with their adapter may simulate composite states in Harel's statecharts without introducing the complexity of the semantics of the composite states. Also, transitions with unions of states diminish the need for composite states. Note that the two approaches to reducing complexity of FSMs can be combined, i.e. Harel's statecharts can be extended with regular expressions of events.

References

[ASU] A. Aho, R. Sethi, J.Ullman Compilers: Principles, Techniques, and Tools. Addison-Wesley, 1985.

[BRJ] G. Booch, J. Rumbaugh, I. Jacobson. Unified Modeling Language User Guide. Addison-Wesley Longman, 1998.

[Cha] D. Chappell. Understanding ActiveX and OLE. Microsoft Press, 1996.

[Cow] J. Cowell. Essential Visual Basic 5.0 Fast. Springer Verlag, 1997.

[EJB] Enterprise JavaBeans™ Specification, v. 1.0. Sun Microsystems, Inc., 1998.

[Eng] R. Englander. Developing Java Beans. O'Reilly & Associates, 1997.

[FO] O. Færgemand and A. Olsen. Introduction to SDL-92, *Computer Networks and ISDN Systems*, Vol. 26, 1994.

[GLJ] S.D. Gray, R. A. Lievano, R. Jennings. Microsoft Transaction Server 2.0. Sams Publishing, 1997.

[Har] D. Harel. Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming*, 1987, #8.

[OMG] OMG Unified Modeling Language Specification, v. 1.3. Object Management Group, Inc., 1999.

[Wie] R. Wieringa. A Survey of Structured and Object-Oriented Software Specification Methods and Techniques. *ACM Computing Surveys*, Vol. 30, 1998, #4.