State Machine Specification Directly in Java and C++

Alexander Sakharov

Problem Statement

Finite state machines (FSM) have become a standard model for representing object behavior. UML incorporates FSM notation that is an object-oriented variant of Harel's statecharts. Specification and code generation facilities are not available in most widely used implementation languages like Java and C++. Availability of FSM-related automation directly in these languages may dramatically increase both the scope of usage of FSM specifications and the productivity of development of a broad range of software. In contrast to development of client-side visual components and server-side transactional components, automation of the development of non-visual non-transactional components, which normally play the role of listeners and which are best modeled by FSMs, has not been much addressed.

Solution for Java and C++

FSMs can be specified directly in procedural object-oriented languages. Besides, FSM transitions are expanded by adding regular expressions of events and by adding unions of states. Assembly of applications out off FSM components can be specified in these languages as well. FSM components are generated from FSM specifications. Threads are generated from assembly specifications. They serve as event adapters, and implement control flow and data interchange between components. Overall, relevant UML's capabilities including concurrency regions and SynchStates are made available in Java and C++ with extra power added by regular expressions of events.

Inheritance is a primary enabler of our approach that is quite generic and applicable to various specifications. An abstract base class declares static variables for storing specifications and contains code generation methods. Derivates of the base constitute FSM specifications. Execution of a compiled specification class does code generation resulting in a class derived from the specification class.

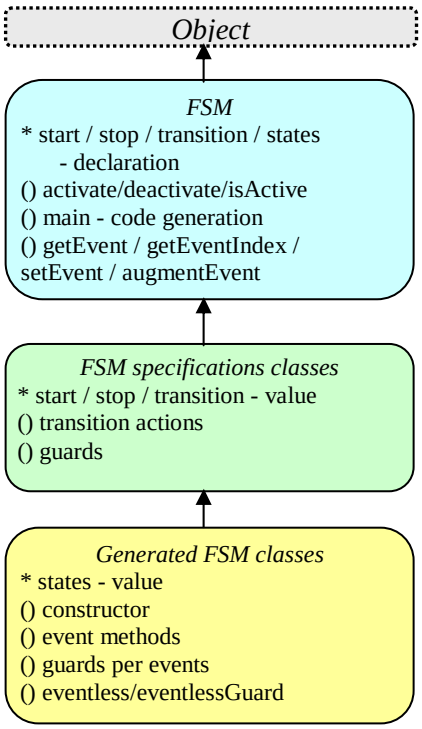
Features and Benefits

- UML's state machines made available in procedural object-oriented languages
- Declarative and procedural styles combined:
  Specifications given in Java or C++ are declarative whereas supporting methods are procedural
- No additional specification languages, no extra tools required
- Generic approach to program specification and code generation directly in Java and C++
- Powerful hybrid notation enriching FSM transitions via
  regular expressions of events and unions of states
- Focus on components:
  Generated components do not need further modification;
  they are serializable, customizable; they may be containers.
- Container components simulate composite states in Harel's statecharts
- UML's concurrent regions supported (including SynchStates)
- Automation of development of non-visual non-transactional listener components
- Complete applications generated from assembly specifications
- FSMs can be programmatically combined with other components, threads, etc.

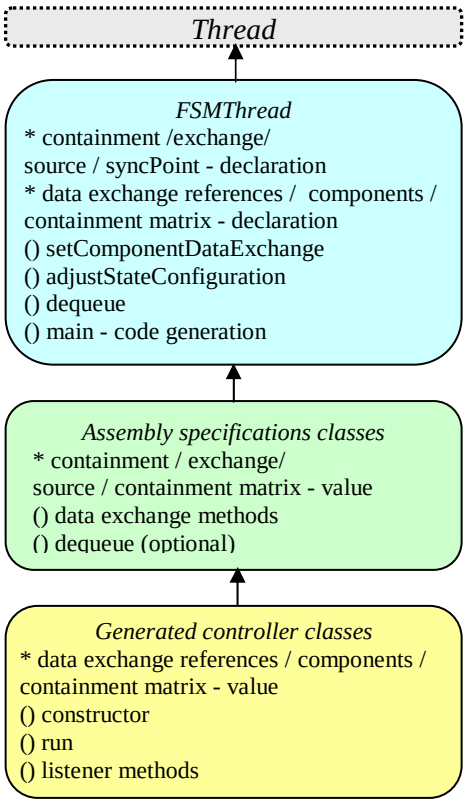
Class Hierarchies

Abstract Class ß *Specification Class* ß *Generated Class*

Components



Controllers



FSM Specification

- Start state name
- Stop state name (optional)
- Transitions
- Action methods and guard code

Sample FSM specification:

```
public class TVHandler extends FSM { ...
    static { ...
        start = “Off”;
        stop = “”;
        transition = new String[][] { { “Off”, “power” “On”, “turnOn” },
            { “On”, “up” “On”, “volume(1)”, "!isError()" },
            { "On", "down", "On", "volume(-1)", "!isError()" },
            ... };
    }
    // transition methods
    public void turnOn() { ... )
    public void volume(double v) { ... }
    ...
    // guards
    public boolean isError() { ... }
    ...
}
```

Hybrid Transition Notation

Transition syntax in UML:

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

We expand conventional FSM transitions with event and state expressions. State expressions may refer to unions of states. Their respective transition specifications denote multiple transitions. The event expression is a regular expression whose tokens are event signatures. Several actions can be given for a transition defined with an event expression. They may relate to transition events or may be ‘default’, ‘normal’, or ‘premature’. The two latter actions are execution upon transition completion.

Our transition syntax:

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

Examples of transitions with state unions and regular expressions of events:

```
(-(Off))+ | ) -> Off [ isError() ] / (:displayError)     // eventless
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(“Channel is not set”) )
incdec : (On|Mute) | (enter (up|down)+) -> .  / (up:incChannel, down:decChannel,
    ~premature:displayWarning(“Channel is not changed”))
```

Component Generation

The classes generated from FSM specifications contain implementations of methods named as events in the transitions specified in their super classes. These generated classes also contain guard methods for the aforementioned transition event methods. Additionally, a couple of methods are generated for handling eventless transitions. Each generated FSM component class has an array with state names. For the sake of efficiency, FSM events and states are represented by integers in generated code. Generated transition event methods and their guards in the sub-classes have the form of switch statements whose cases are FSM states including pairs with hidden states. Branches in the switch statements of the guard methods test guard expressions. Every branch in the switch statements of the transition event methods calls an applicable transition action whose guard condition is satisfied and updates the FSM state.

Generated FSM component:

```
public class TVSpecificationSub extends TVSpecification { ...
    public static String[] state = { ... };
    protected Vector incdec = new Vector();
    ...
    public int enter( EventObject evt ) { ... switch (stateIndex) { ...
        if ( transitionIndex==n ) { action(); stateIndex = m; break; }
    ... } ... }
     ...
    public int eventless() { ... switch (stateIndex) { ... } ... }
    public boolean enterGuard( EventObject evt ) { ... switch (stateIndex) { ...
        if ( ... ) { transitionIndex = n; return true; }
    ... } ... }
    ...
    public boolean eventlessGuard() { switch (stateIndex) { ... } } ...
}
```

Internal FSMs

Transitions with regular expressions of events containing Kleene closure or concatenation are turned into internal FSMs augmenting their host FSMs. These generated internal FSMs are minimized after that. The internal FSMs add hidden states to the set of states of the host FSM. The 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 respective regular expression. Transitions with event expressions containing unions only are unfolded into multiple conventional transitions.

Example illustrating generated method enter:

```
void enter(EventObject eventValue) {
    switch (stateIndex) {
        case <On> :
            incdec.addElement(eventValue); stateIndex = <On,Incdec:1> ; break;
        case <On,Channel:1> :            // hidden state
            channel. addElement(eventValue);  enterChannel(); stateIndex = <On,Channel:2> ; break;
        case <On,Channel:2> :            // hidden state
            channel.removeAllElements();
            stateIndex = <On > ; enter(eventValue); break;
        case <On,Incdec:1> :             // hidden state
            displayWarning("Channel is not changed")); incdec.removeAllElements();
            stateIndex = <On > ; enter(eventValue); break;
        case <On,Incdec:2> :             // hidden state
            incdec.removeAllElements();
            stateIndex = <On > ; enter(eventValue); break;
        case … // other states – code for Mute is similar to above
        default: break;
} }
```

Assembly Specification

- **Containment**
  Specified by list of { container component, container state, sub-component }
  Multiple components mapped to the same (component, state) pair are run concurrently
  This relation extends onto relevant hidden states
- **Data exchange** methods for component pairs
  Specified by list of { container component, sub-component, method1, method2, flag }
  The flag indicates whether data exchange happens at invocation and at reaching a stop state, or at invocation and at every termination of the sub-component.
- **Synchronization points** for concurrent regions (a la SynchState in UML)
  Specified by list of { component1, component2, state/transition, state/transition }
- **Event sources and mapping** of listener interface methods to transition events
  Specified by list of { component, source, transition event, listener interface method }
- **Event delivery mode**
  (synchronous, asynchronous without queuing, or asynchronous with queuing)

Sample assembly specifications:

```
public class TVApplication extends FSMThread { ...
    static { ...
        containment = new String[][] { { “TVHandler”, “On”, TVSet” }, { “TVHandler”, “On”, TVDevice.cd” }, ... );
        exchange = new String[][] { { “TVHandler”, TVSet”, “inSet”, “outSet” },
            { “TVHandler”, TVDevice.cd” “inDevice”, outDevice”}, ... );
        source = new String[][] { { “TVHandler”, “TVRemote.rc”, “power”, “power”, “enter”, “enter”, ... },
            { TVSet” “TVRemote.rc”, “toggle”, “swap”, ... } ),
            { “TVDevice.cd”, “TVRemote.rc” “init”, “device”, “exec”, “init”, ... } );
        syncPoint = new String[][]
            { { “TVSet”, “TVDevice.cd”, “stateSet”, “transitionDevice”, “transitionSet”, “stateDevice”, ... }, ... };
        delivery = ‘s’;
    }
    // data exchange methods
    public static void inSet(TVHandler h, TVSet s) { ... )
    public static void outSet(TVHandler h, TVSet s) { ... }
    ...
}
```

Application Generation

The controllers generated from assembly specifications are threads that implement control flow between containers and their sub-components and play the role of event adapters for all relevant FSM components. The generated controllers implement all listener interface methods for specified event sources. Each generated controller has a static Boolean array employed for fast determination of containment and another static array with references to data exchange methods. These controllers construct instances of generated FSM component classes. They construct and fire event sources, and then register themselves as listeners to the event sources. The controllers queue events when so specified. The controllers call counterparts of listener interface methods and 'eventless' methods of the classes generated from FSM specifications. The default order of processing events in the queue is FIFO, which can be overridden.

Generated controller:

```
public class TVAppAdapter extends TVApp implements Runnable, EventListener, EventSource1, ... {
    public TVAppAdapter() { ... } // sets data exchange
    private Vector eventQueue; private Vector eventIndexQueue;
    private static EventSource source[] = { ... };
    static { component = new FSM[] { ... };
    isSubComponent = new boolean[][][] { ... }; }
    private Thread sourceThread[];
    protected int dequeue() { ... } // to override FIFO
    ...
}
```

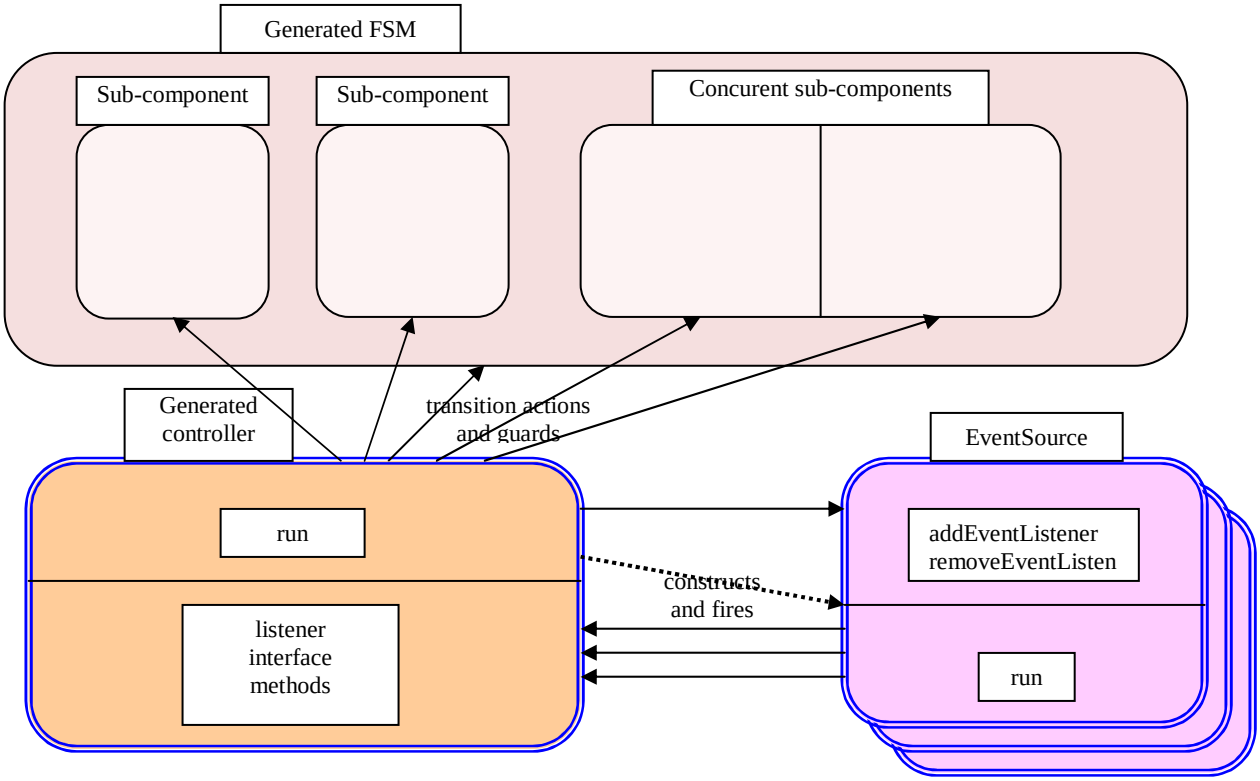
Event procedures in generated controllers:

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

The core of method run of generated controller:

```
for (;;) { …
    switch( eventIndex ) { …
        case <foo > :
            iterator = active.iterator();
            label: while ( (point = (StateConfiguration)iterator.postorder())!=null ) {
                switch ( point.getComponent() ) {
                    case <componentx> :
                        ((ComponentxSub)component[ <componentx> ]).setEvent(event, eventIndex);
                        if ( ((ComponentxSub)component[ <componentx> ]).fooGuard(eventValue) ) {
                            stateIndex = ((ComponentxSub)component[ <componentx> ]).foo(eventValue);
                            adjustStateConfiguration(point, stateIndex);
                            break label;
                        }
                        break;
                        …
} } } } } }
```

Call Graph


Generated FSM
Sub-component
Sub-component
Concurent sub-components
Generated controller
transition actions and guards
EventSource
run
addEventListener
removeEventListen
constructs and fires
listener interface methods
run