State-Oriented Programming

Asher Sterkin

NDS Technologies Israel Ltd.
P.O. Box 23012, Har Hotzvim
Jeruslaem 91235 Israel
asterkin@nds.com

Abstract. This article discusses the differences between traditional Object-Oriented vs. State-Oriented programming paradigms. In particular, it stresses the need to effectively combine both paradigms. In the article, we will discuss multiple options for mapping between statechart elements and an Object-Oriented programming language elements. We will then suggest a practical way of embedding UML 2.0 statechart formalism in the form of Domain-Specific Language directly into a general purpose OO programming language (Groovy). Concrete implementation of a subset of UML 2.0 statechart elements, called Generic Statechart Library, is presented.

1 Introduction

One of the main advantages of the Object-Oriented Programming paradigm is that it provides a cost-effective way to get rid of complex conditional logic through extensive usage of encapsulation and polymorphism. Each object is supposed to "know" its type, to keep its state internally and to respond accordingly to external stimuli. Object-Oriented programming is conceptually different not only from procedural programming but also from functional programming. The latter also supports polymorphism extensively, but conceptually precludes *any* side effects including stateful objects [3]. The main reasoning behind this restriction is that without side effects, it's much easier to ensure software correctness and reliability. As the recent trend claims, side effects are also a major obstacle toward software scalability, especially in designing many multi-core software [4].

A very popular architectural solution applied to building highly-scalable data processing and/or real-time systems is the so-called *Active Object* design pattern [16, 5, 9] where any side effects are limited to the internals of active objects which work simultaneously but communicate with one another sequentially via message queues. As long as the active object's (or some of its internals) behavior depends solely on its type, it will work perfectly. The problem starts when incoming message processing also depends on the object's current state.

The traditional Object-Oriented approach would suggest extracting this state-dependant behavior into one or more separate objects and delegating to them decisions about which particular operations need to be performed on the main object (see *State* design pattern in [16]. This solution, however, does not scale

well for a large number of potential states/transitions. The same happens when an Object-Oriented solution for plain Final State Machines (FSMs) is proposed: it is not scalable for a large number of states and thus is impractical to be used unless the FSM is built automatically from some other formal description, as usually happens with regular expressions and formal grammar [6].

Statecharts [7, 8] suggest a scalable solution for state-dependent system modeling and specification. The Statechart formalism significantly reduces both the number of states and transitions by using hierarchical and parallel states, special *onEntry* and *onExit* event handlers, shallow and deep history. The Object Management Group (OMG) adopted the Statechart formalism for object behavior modeling and included it as part of the Unified Modeling Language (UML) specification [8].

Originally, Statecharts were associated with *big money* complex CAD tools such as iLogic Rhapsody or IBM/Rational Rose Real-Time. As such, they were disliked by many developers due to an excessive license cost and a special coding style generated by these tools. However, the main problem with these tools is probably not the license cost, since even free tools are not always popular. The problem is the huge gap between statechart diagrams and the code generated from them. This does not occur with class diagrams where a close correspondence between graphical and textual representation is easily recognizable for most popular programming languages. Therefore, a new programming paradigm, let's call it State-Oriented Programming, is required in order to introduce statecharts in mainstream software development. In order to make it practical, the State-Oriented Programming paradigm should come as an extension of the established Object-Oriented Programming paradigm. More specifically, any prospective State-Oriented Programming language should come as a Domain-Specific Language (DSL) [2] embedded directly in an Object-Oriented programming language such as Java, Groovy, Ruby, etc.

In order to achieve a workable solution, multiple tradeoffs need to be properly balanced, for example, source code readability, correct UML semantics implementation, CPU efficiency, memory consumption and footprint.

In more general terms, embedding a Statechart Domain Specific Language (DSL) within a General Object-Oriented Programming Language (GOPL) turns out to be a complex mapping problem where various elements of the Statechart formalism (states, transitions, guards, actions) need to be mapped onto different elements of the hosting programming language (classes, methods, fields, templates, annotations and aspects).

This article presents a Generic Statechart Library (GSL) implemented in Groovy [13]. GSL provides a cost-effective implementation of an essential subset of Statechart formalism, suitable for user interface intensive and communication systems. Special attention is paid to code readability and trying to make GSL constructs look as close as possible to Groovy.

2 State Programming: historical background

The first time the idea of using the Statechart mechanism directly in a General Programming Language (GPL) was realized was in the Quantum Event Processor (QEP) framework [9], which supports hierarchical states in C/C++. Later, a Boost Statechart Library (BSL) with full UML 2.0 Statechart support via C++ templates was developed as part of the Boost project [10]. The W3C consortium issued an XML schema for Statechart specification, called SCXML [11], and Apache Group developed a Commons SCXML library, which supports SCXML interpretation in Java [12]. All this demonstrates a growing interest in the use of Statecharts by software developers.

Still, the question of whether to support the Statechart programming paradigm directly within modern Object Oriented Programming languages such as Java, Groovy, Ruby, etc. remains open.

3 State-Oriented Programming in Java. Is it practical?

3.1 First Simple Example

Perhaps the simplest possible statechart diagram consists of two states as presented in the diagram below.

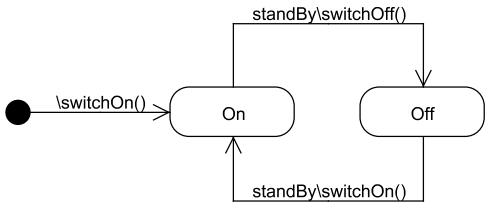

Fig. 1. Simple Statechart: Toggle Behavior

This kind of behavior is very common for all devices with a toggle button: TV, STB, Handheld computers, and mobile phones. Some GUI applications also use this behavior extensively. The main reason for using toggles is the lack of space: we would like to minimize the number of buttons and thus utilize the same button for mutually exclusive operations. Which operation to invoke depends on the current state: switch on if it's off and vice versa.

The most straightforward implementation of toggle behavior would be to use a single boolean variable reflecting the current state. For example:

```
public class DeviceController1 {
  private boolean isOn = false;
```

```
  void switchOn()   {/*do something*/}
  void switchOff()  {/*do opposite*/ }

  public void powerUp() {
    switchOn();
    isOn = true;
  }
  public void standBy() {
    if(isOn)
      switchOff();
    else
      switchOn();
    isOn = !isOn;
  }
}
```

This solution would be perfectly OK for such a small exercise, however it does not scale well with the number of possible states and events. In any realistic application, the number of state variables and conditional logic operations would be enormous. Object Oriented programming favors polymorphism over variables for implementing complex conditional logic.

3.2 Using "State" Design Pattern

A straightforward Object-Oriented solution following the "Gang Of Four" book State design pattern [16] would look as follows:

```
public class DeviceController2 {
  static class State {
    void powerUp(final DeviceController2 context) {}
    void standBy(final DeviceController2 context) {}
  }
  static class Initial extends State {
    private static State instance = null;

    void powerUp(final DeviceController2 context) {
      context.switchOn();
      context.setState(On.getInstance());
    }
    static State getInstance() {
      if(null == instance)
        instance = new Initial();
      return instance;
    }
  }
  static class On extends State {
```

```
    private static State instance = null;

    void standBy(final DeviceController2 context) {
      context.switchOff();
      context.setState(Off.getInstance());
    }
    static State getInstance() {
      if(null == instance)
        instance = new On();
      return instance;
    }
  }
  static class Off extends State {
    private static State instance = null;

    void standBy(final DeviceController2 context) {
      context.switchOn();
      context.setState(On.getInstance());
    }
    static State getInstance() {
      if(null == instance)
        instance = new Off();
      return instance;
    }
  }
  private State state = Initial.getInstance();

  public void powerUp() {
    state.powerUp(this);
  }
  public void standBy() {
    state.standBy(this);
  }
  void setState(final State state) {
    this.state = state;
  }
  void switchOn() {
    //do something
  }
  void switchOff() {
    //do opposite
  }
}
```

This is not the whole story. Events are seldom processed via direct method invocation and too often need to first be mapped from some string or integer

value. For this purpose, the Command design pattern [16] is usually applied as follows:

```
import java.util.HashMap;
import java.util.Map;

public class CommandRepository2 {
  abstract class Command {
    abstract void execute();
  }
  final Map<String, Command> commandMap =
    new HashMap<String,Command>();
  final DeviceController2 context;

  public CommandRepository2(final DeviceController2 ctx) {
    this.context = ctx;
    commandMap.put("powerUp", new Command(){
      void execute(){context.powerUp();}});
    commandMap.put("standBy", new Command(){
      void execute(){context.standBy();}});
  }
  public void execute(final String event) {
    commandMap.get(event).execute();
  }
}
```

We actually get an unpleasant surprise: This pure OO solution is much more verbose than its state variable counterpart. Even worse, the OO version looks very different from the original statechart diagram. As with CAD tools mentioned above, the main problem is the huge gap between the diagram and its corresponding code.

3.3 Using Java Anonymous Classes and Reflection

A more compact, though a bit less of an OO solution, could look as follows:

```
import java.lang.reflect.Method;

public class DeviceController3 {
  public void handleEvent(final String event) throws Exception {
    final Method method = state.getClass()
            .getDeclaredMethod(event, new Class[]{});
    method.invoke(state, new Object[]{});
  }
  void switchOn()   {/*do something*/}
  void switchOff()  {/*do opposite*/ }
```

```
  class State {}

  private State state = new State() {
    void powerUp() {switchOn();  state=ON;}
  };
  private final State ON = new State() {
    void standBy() {switchOff(); state=OFF;}
  };
  private final State OFF   = new State() {
    void standBy() {switchOn();  state=ON;}
  };
}
```

The proposed alternative looks pretty close to the original statechart: each state is represented by a separate anonymous class, and each event is represented by a separate method. In principle, this is the same as it was in the previous solution, but due to its terse form the Java code delivers the intent more directly. Another crucial difference is that in that proposal, events are not implemented in a polymorphic way, but rather are invoked through Java reflection.

The main problem with declaring all possible events in the base State class is that the event/state matrix is usually very sparse: very seldom all events are handled at all states. Declaring all possible events upfront in the base State class could easily obscure the original statechart. It also eliminates a need for elaborated mapping between external event representation (e.g. String) and event methods.

The proposed method will not work, however, if events are encoded in non-string format, for example as numbers.

Another potential issue with this solution is that we will explicitly create a separate State object for each context. While for simple UI applications it is not a problem, it might be too wasteful for high-throughput communication software. Conceptually, State is supposed to be stateless, e.g., not hold any data. This is considered as an important "divide and concur" principle of the State pattern: the context holds data and knows how to manipulate it, the state knows when which operation has to be performed. This drawback could be overcome by using Java enumerations as follows:

```
import java.lang.reflect.Method;

public class DeviceController4 {
  public void handleEvent(final String event) throws Exception {
    final Method method = state.getClass()
        .getDeclaredMethod(event
            ,new Class[]{DeviceController4.class});
    method.invoke(state, new Object[]{this});
  }
```

```
  void switchOn()   {/*do something*/}
  void switchOff()  {/*do opposite*/ }

  enum State {
    INITIAL() {
      void powerUp(final DeviceController4 ctx) {
        ctx.switchOn(); ctx.setState(ON);
      }
    },
    ON() {
      void standBy(final DeviceController4 ctx) {
        ctx.switchOff(); ctx.setState(OFF);
      }
    },
    OFF() {
      void standBy(final DeviceController4 ctx) {
        ctx.switchOn(); ctx.setState(ON);
      }
    }
  }
  State state = State.INITIAL;
  void setState(final State state) {
    this.state = state;
  }
}
```

3.4 Supporting onEntry()/onExit() Handlers

In the original statechart, the *switchOn()* operation is duplicated between two transitions: from INITIAL to ON and from OFF to ON. Using the statechart *onEntry/ onExit* mechanism would lead to a more compact solution:

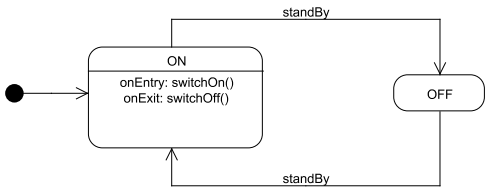

Fig. 2. Simple Statechart with onEntry/onExit

The corresponding Java implementation will look like:

```
import java.lang.reflect.Method;
```

```
public class DeviceController5  {
  public void handleEvent(final String event) throws Exception {
    final Method method = state.getClass()
        .getDeclaredMethod(event
            ,new Class[]{DeviceController5.class});
    method.invoke(state, new Object[]{this});
  }
  void switchOn()   {/*do something*/}
  void switchOff()  {/*do opposite*/ }

  enum State {
    INITIAL() {
      void init(final DeviceController5 ctx) {
        ctx.setState(ON);
      }
    },
    ON() {
      void onEntry(final DeviceController5 ctx) {
        ctx.switchOn();
      }
      void onExit(final DeviceController5 ctx) {
        ctx.switchOff();
      }
      void standBy(final DeviceController5 ctx) {
        ctx.setState(OFF);
      }
    },
    OFF() {
      void standBy(final DeviceController5 ctx) {
        ctx.setState(ON);
      }
    };
    void onEntry(final DeviceController5 ctx){/* nothing by default*/}
    void onExit(final DeviceController5 ctx) {/* nothing by default*/}
  }
  State state = State.INITIAL;

  void setState(final State state) {
    this.state.onExit(this);
    this.state = state;
    this.state.onEntry(this);
  }
}
```

3.5 Supporting Nested States

The method proposed above will work pretty well for relatively simple statecharts. However, it does not scale up when nested states need to be implemented. For example, imagine that when the device (for example, digital TV Set Top Box) is switched ON, it should process a special key "+"in order to show on/off a banner with some useful information about the current TV Channel and program:

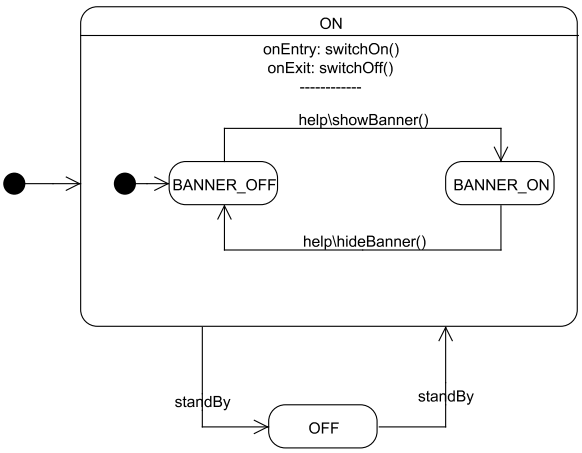

Fig. 3. Simple Statechart with nested states

Even this simple nested statechart would enforce us to abandon *onEntry* and *onExit* mechanisms in this implementation, thus reverting to substantial code duplication.

```
import java.lang.reflect.Method;

public class DeviceController6 {
  public void handleEvent(final String event) throws Exception {
    final Method method = state.getClass()
    .getMethod(event
        ,new Class[]{DeviceController6.class});
    method.invoke(state, new Object[]{this});
  }

  void switchOn()   {/*do something*/}
  void switchOff()  {/*do opposite*/ }
  void showBanner() {/*do something*/}
  void hideBanner() {/*do opposite*/ }
```

```
  static class State {
    State init() {return null;}
  }
  static final State INITIAL = new State() {
    public void start(final DeviceController6 ctx) {
      ctx.switchOn(); ctx.setState(ON);
    }
  };
  static class State_ON extends State {
    State init() { return BANNER_OFF; }
    public void standBy(final DeviceController6 ctx) {
      ctx.switchOff(); ctx.setState(OFF);
    }
  }
  static final State ON = new State_ON();
  static final State BANNER_OFF = new State_ON() {
    State init() { return null;}
    public void help(final DeviceController6 ctx) {
      ctx.showBanner(); ctx.setState(BANNER_ON);
    }
  };
  static final State BANNER_ON = new State_ON() {
    State init() { return null;}
    public void help(final DeviceController6 ctx) {
      ctx.hideBanner(); ctx.setState(BANNER_OFF);
    }
  };
  static final State OFF = new State() {
    public void standBy(final DeviceController6 ctx) {
      ctx.switchOn(); ctx.setState(ON);
    }
  };

  State state = INITIAL;

  void setState(State state) {
    State s = state.init();
    while(null != s){
      state = s; s = state.init();
    }
    this.state = state;
  }
}
```

We have again came up with a solution which is very verbose and looks too far from the original statechart.

Although inheritance is the most natural way for implementing nested states and indeed could be used for implementing simple models, it comes at odds with *onEntry*, *onExit* mechanism, at least in programming languages which do not have explicit destructors. These two *onEntry()/onExit()* functions cannot be automatically inherited, but rather a decision which to invoke should be made using the "Least Common Ancestor" [9] algorithm. Apparently, the Statechart and Object-Oriented run-time models are not compatible: they have some fundamental differences.

Also, Java seems to be not very meta-programming [2] friendly and enforces us to put quite a lot of extra code (called "noise") in order to get even this trivial model to work. Extracting even a simple framework from this code would be almost impossible due to some Java security limitations.

3.6 Preliminary Conclusions

Statecharts need to be treated as a Domain Specific Language embedded in a General Purpose Programming Language (in this case Java)[2]. When developing an embedded DSL, we have to map its elements onto the *host* programming language elements. For example, we need to decide how to represent simple, nested and parallel states, transitions, guards, and special nodes: initial, choice, junction, etc. When using Java, we could map them onto named or anonymous classes, objects, methods, annotations, aspects, etc. The main criteria for selecting this or another mapping should be code readability, efficiency, and sometimes portability.

By applying the GOF State pattern, we tried to map simple and nested states onto anonymous classes and map transitions directly to methods. Very soon, we discovered that this direct mapping is very limiting: we cannot effectively implement *onEntry*, *onExit* and we do not have even the slightest idea about how to implement guards rather than through plain if-then-else. We still have a very long way to implementing the full set of UML 2.1 Statechart features.

3.7 Alternative Mapping

It seems that the core problem is with suboptimal mapping between statecharts and Java elements. Mapping states onto anonymous Java classes looks like a useful idea. The same is with mapping special events such as *onEntry* and *onExit* onto Java class methods. The problem seems to be with implementing nested States via inheritance and implementing transitions as methods. Let's see if we could somehow change these two mappings (for nested states and transitions) in order to get better solution.

One possible alternative is to use object composition for implementing nested states and representing transitions as objects. Here is an implementation of the previous Statechart using this new approach:

```
import org.gsl.java.*;

public class DeviceController7 extends StateMachine {
  private final CompositeState ON = new CompositeState() {
    protected Vertex initial() { return BANNER_OFF; }
    protected void   onEntry() { switchOn(); }
    protected void   onExit()  { switchOff();}

    protected Transition[] transitions() {
      return new Transition[] {
        new Transition() {
          protected boolean guard() { return "standBy".equals(event); }
          protected Vertex  target(){ return OFF; }
        }
      };
    }
  };

  private final State OFF = new SimpleState() {
    protected Transition[] transitions() {
      return new Transition[] {
        new Transition() {
          protected boolean guard() { return "standBy".equals(event); }
          protected Vertex  target(){ return ON; }
        }
      };
    }
  };

  private final State BANNER_OFF = new SimpleState(ON) {
    protected Transition[] transitions() {
      return new Transition[] {
        new Transition() {
          protected boolean guard() { return "help".equals(event); }
          protected Vertex  target(){ return BANNER_ON; }
          protected void    action(){ showBanner(); }
        }
      };
    }
  };

  private final State BANNER_ON = new SimpleState(ON) {
    protected Transition[] transitions() {
      return new Transition[] {
        new Transition() {
```

```
          protected boolean guard() { return "help".equals(event); }
          protected Vertex  target(){ return BANNER_OFF; }
          protected void    action(){ hideBanner(); }
        }
      };
    }
  };

  void switchOn()   {/*do something*/}
  void switchOff()  {/*do opposite*/ }
  void showBanner() {/*do something*/}
  void hideBanner() {/*do opposite*/ }

  protected Vertex initial() { return ON; }

  private String event = null;
  public void handleEvent(final String event) {
    if("init".equals(event))
      super.init();
    else {
      this.event = event;
      super.handleEvent();
    }
  }
}
```

In the proposed method, the decision was made to give up on the event lookup algorithm and to stick with a generalized *guard()* method. This *guard()* method is internally invoked by framework for each transition in order to find the first transition, which is enabled. Defining and supporting a general-purpose Event class could unnecessarily complicate the framework without any significant benefit. Surprisingly, it's easier to store event information outside the statechart framework and to rely on inner classes access to its content.

This approach was implemented in a form of Generic Statechart Library (GSL), which supports all main UML Statechart constructs in a cost-effective way: nested and parallel states, dynamic choices and actions. The first version of GSL was implemented in Java as an initial proof-of-concept that State-Oriented Programming is possible in principle.

However, many useful ideas needed to be rejected: annotations, reflection and deeply nested classes as potential means to naturally reflect nested state hierarchy. Neither of these options lead to a workable solution and after spending some time on experimenting, we were forced to give up. Apparently, Java is quite resistant to advanced meta-programming. Something like the code fragment presented below just will not work in Java:

```
public class DeviceController8 extends StateMachine {
```

```
  @Init public final State ON = new State() {
    protected void   onEntry() { switchOn(); }
    protected void   onExit()  { switchOff();}

    protected Transition standBy = new Transition(OFF);

    @Init public final State BANNER_OFF = State() {
      protected Transition help = new Transition(BANNER_ON) {
        protected void action() { showBanner(); }
      }
    };

    public final State BANNER_ON = State() {
      protected Transition help = new Transition(BANNER_OFF) {
        protected void action() { hideBanner(); }
      }
    };
  };

  public final State OFF = new State() {
    protected Transition standBy = new Transition(OFF);
  };
  void switchOn()   {/*do something*/}
  void switchOff()  {/*do opposite*/ }
  void showBanner() {/*do something*/}
  void hideBanner() {/*do opposite*/ }
}
```

In order to achieve this or an even higher level of code clarity, we shall turn our attention to the next generation scripting languages: JRuby, Groovy, and Scalla.

4 State-Oriented Programming in Groovy

Even the most recent Java 6 lacks some ingredients crucially important for implementing a fluent embedded Domain-Specific language like what we are trying to achieve: closures, relaxed syntax, and advanced meta-programming. All these features are supported out of the box in the next generation of Object-Oriented Programming languages: Groovy[13], Ruby[14], Scala[15], etc. Sometimes these languages are called *dynamic*, reflecting the fact that these languages are effectively Object-Oriented scripts not too different in this regard from Perl, Tcl, or Python. The difference is a cleaner syntax supporting both Object-Oriented and functional programming and good integration with Java Virtual machine.

Actually, all three programming languages (Groovy, Ruby [1], Scala) come as a natural extension of the Java programming model.

4.1 Why Groovy?

In the quest for State-Oriented Programming DSL the Groovy programming language was chosen as the first candidate for a very basic reason: its syntax is very close to that of Java, thus minimizing a potential learning curve. The same feature set could be implemented in JRuby in a straightforward manner. Scala suggests some interesting additional features such as a smooth integration between Object-Oriented and functional programming, including good support for concurrent programming comparable to that of Erlang [4]. It will be more closely evaluated for the next version of the GSL, especially as a candidate for cost-effective implementation of parallel states[7] and potential performance improvements. The main goal for now is to achieve a high level of code clarity, and Groovy serves this purpose pretty well.

The subsequent sections of this paper describe major elements of the Groovy version of the Generic Statechart Library (GSL-G) in greater detail.

4.2 Sample Statechart Revised

We will start with a slightly more complicated variation of the sample Statechart diagram we used in the previous section (see Fig. 4).

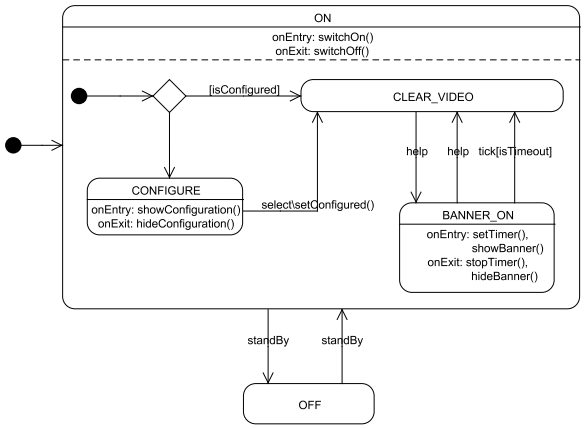

Fig. 4. Electronic Program Guide: Banner and Setup

This statechart describes a tiny fraction of the functionality of a hypothetical Electronic Program Guide (EPG) GUI application which usually resides in

[1] In the case of Ruby, it would be the so-called JRuby - a Java implementation of the Ruby programming language. For more details, go to http://jruby.codehaus.org/

a digital TV Set- Top Box. It describes a very small subset of EPG functionality, yet it is still challenging enough to be implemented using a conventional Object-Oriented method. The question is not whether it is possible to implement sophisticated GUI functionality just in plain Java or even "C". The fact is that such applications are implemented. The question is rather how quickly programmer will lose control over the application behavior due to unmanageable complexity. In our experience, lack of a proper statechart model directly supported in a programming language is the number one reason for so many UI inconsistencies and extra features: very soon it starts to get so complex that nobody dares to make any changes except for fixing the most severe bugs or introducing a completely new functionality such as support for the Personal Video Recording (PVR) functionality. Usually after the change is made, it triggers a new vicious circle of unmanageable complexity and the situation just gets worse until somebody manages to convince upper management to rewrite the application from scratch. This is hardly a sustainable business model.

4.3 Testing Statecharts

Before we take a look at the corresponding Groovy code, let's ask a very simple question: how can we test statecharts? Indeed, the overall number of potential scenarios grows exponentially in statecharts of any realistic application. Running through them all is neither practical nor physically possible. Even worse, tests of many GUI applications are performed either manually or through a GUI runner automation tool such as Mercury WinRnner. This is not a practical approach and having good support for statechart testing must be made a top priority requirement for any prospective tool or library.

In the case of GSL, we made a decision to conduct statechart testing by running a short representative scenario (roughly corresponding to agile user stories [17]) using an open source Framework for Integrated Tests (FIT) library [18].

More specifically, each test is represented as a Column Fixture [18] table where the first input column represents a list of incoming event. One or more additional input columns represent various guards, and the last column represents a list of actions being performed.

Table 1. EPG: Basic StandBy Acceptance Test

epg.TestUI		
Events	isConfigured	Actions?
start	true	switchOn
standBy	n/a	switchOff
standBy	true	switchOn

The basic StandBy behavior test is presented in Table 1. This particular test scenario merely states that when a STB is configured, nothing specific happens during power up (represented as a special "start" event) except for switching

video on. It also states that video is switched off/on each time the "standBy" button is pressed.

Table 2. EPG: Basic Configuration Acceptance Test

epg.TestUI		
Events	isConfigured	Actions?
start	false	switchOn, showSetup
standBy	n/a	hideSetup, switchOff
standBy	false	switchOn,showSetup
select	n/a	hideSetup, setConfigured
start	true	switchOn

The basic configuration functionality test is presented in Table 2. The main lesson we need to learn here is that contrary to the common belief, Statecharts are mainly a *design* rather than a *requirements* specification tool: the same acceptance test scenario could be correctly realized by multiple statecharts varying in size and expressive power. In our experience, building a good statechart is a skill, which improves with experience, but can hardly be formalized as a strict engineering discipline.

The test scenario presented above could be implemented using a simple Groovy script, as follows:

```
package epg

import fit.ColumnFixture

class TestUI extends ColumnFixture {
  public  def String       Event
  public  def String       isTimeout
  public  def String       isConfigured
  private def              actions
  private def UIController controller
  private def guardMap = ['true':true, 'false':false]

  TestUI()
  {
    controller = new UIController()
    controller.switchOn      = {actions << 'switchOn'}
    controller.switchOff     = {actions << 'switchOff'}
    controller.showBanner    = {actions << 'showBanner'}
    controller.startTimer    = {actions << 'startTimer'}
    controller.stopTimer     = {actions << 'stopTimer'}
    controller.isTimeout     = {guardMap[isTimeout]}
    controller.hideBanner    = {actions << 'hideBanner'}
```

```
    controller.isConfigured  = {guardMap[isConfigured]}
    controller.showSetup     = {actions << 'showSetup'}
    controller.hideSetup     = {actions << 'hideSetup'}
    controller.setConfigured = {actions << 'setConfigured'}
  }

  String[] Actions() {
    return actions
  }

  void execute() {
    actions = []
    if('start'==Event)
      controller.start()
    else
      controller.handleEvent(Event)
  }
}
```

Here, the main advantage of a JVM-based scripting language such as Groovy appears: it could be directly used with the FIT library written in Java.

Describing all technical details of the script is beyond the scope of this article. What is important is that it creates an instance of a class under test (called UIController), intercepts some functionality (in this case using the Groovy closures mechanism[13]) and invokes the UIController *start()* and *handleEvent()* methods. That's it. The GSL library was explicitly designed to support test automation through clear isolation of the state-dependent logic from the rest of the system and defining a very narrow yet generic interface for sending events.

4.4 Statecharts Specification in GSL

Now we are ready to take a look at the source of our tiny Electronic Program Guide UIControler:

```
package epg
import org.gsl.groovy.*

class UIController extends StateMachine {
  void define() {
    initial state('ON') {
      onEntry switchOn
      onExit  switchOff

      on('standBy').to('OFF')

      initial choice {
```

```
        guard(isConfigured).to('CLEAR_VIDEO')
        to('CONFIGURE')
      }
      state('CLEAR_VIDEO') {
        on('help').to('BANNER_ON')
      }
      state('BANNER_ON') {
        onEntry startTimer, showBanner
        onExit  stopTimer , hideBanner

        on('help').to('CLEAR_VIDEO')
        on('tick').guard(isTimeout).to('CLEAR_VIDEO')
      }
      state('CONFIGURE') {
        onEntry showSetup
        onExit  hideSetup

        on('select').action(setConfigured).to('CLEAR_VIDEO')
      }
    }
    state('OFF') {
      on('standBy').to('ON')
    }
  }

  def  switchOn      = {/*do something real*/}
  def  switchOff     = {/*do something real*/}
  def  showBanner    = {/*do something real*/}
  def  startTimer    = {/*do something real*/}
  def  stopTimer     = {/*do something real*/}
  def  isTimeout     = {true}
  def  hideBanner    = {/*do something real*/}
  def  isConfigured  = {true}
  def  showSetup     = {/*do something real*/}
  def  hideSetup     = {/*do something real*/}
  def  setConfigured = {/*do something real*/}
}
```

The script is fairly simple, is much shorter than than any alternative encoded in plain Java, and, most importantly, is very close to the original statechart. This script demonstrates all major GLS elements: state machine, states including nested, initial and dynamic choice pseudo-states, transitions, guards, actions onEntry and onExit, special handlers.

Each element is described in more detail in the following subsections.

4.5 Specifying State Machine

In GSL, state machines are primarily designed for implementing the Controller part of the Model-View-Controller design pattern[5]. In our experience, this is where the vast majority of state-dependent behavior is located. The UML class diagram on Fig.5 illustrates the intent.

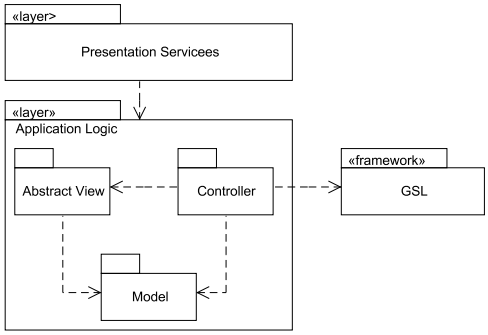

Fig. 5. State Machine as a Controller of the MVC pattern

This architectural decision highlights an important interaction between Object-Oriented and State-Oriented programming paradigms: while the Object-Oriented programming paradigm is fine tuned for programming *entities*, the State-Oriented paradigm is fine tuned for programming *processes*.

In GSL, state machines are specified by deriving from the org.gsl.groovy.StateMachine abstract base class. The state machine specification is supplied in the body of a special *define()* method (resonating with the Groovy **def** statement):

```
class MyStateMachine extends StateMachine {
  void define() {
//the state machine definition goes here
  }
}
```

In continuation with an established Groovy (and Ruby) tradition, GSL applies the Builder[16] design pattern for state machine construction. More specifically, every state machine element (state, transition, etc.) is defined by calling a GSL function with the same name in the scope of *define()* function. State machine elements used in the Electronic Program Guide sample are described in the subsequent sections below.

4.6 Specifying States

In GSL, states are specified by calling a *state()* function within the scope of *define()* function:

```
void define() {
  state('state label goes here') {
    //state specification goes here
  }
}
```

Ideally, it should be possible to avoid function syntax and string constants altogether and to use the Groovy relaxed syntax[13] thus making it look like an almost natural state definition clause:

```
void define() {
  state state-label {
    //state specification goes here
  }
}
```

For a number of technical reasons, it was not accomplished in the current version of GSL and is planned for improvement in the next version.

State definition inside the curly braces could contain *onEntry*, *onExit*, *initial* and *transition* statements. As mentioned above, there is an outmost state automatically created for the state machine for which all these elements could be defined at the top level scope.

4.7 Specifying Initial Pseudo-state

In GSL, the initial pseudo-state[8] is specified as an *inital* prefix for some state. In accordance with the UML specification, there should be only one *initial* statement for each composite state, including the state machine itself:

```
void define() {
  initial state('state1') { //initial state for the state machine
    initial state('sub-state2') { //initial sub-state for state1
      //sub-state specification goes here
    }
    //state specification goes here
  }
}
```

All this might look like magic to those who are unfamiliar with modern scripting language meta-programming capabilities. For those who are familiar, there is nothing magical here: *initial* is a function taking a reference to state object as an argument. Parenthesis could be omitted due to the Groovy relaxed syntax convention, which makes this definition more readable and similar to elements of the host programming language itself.

4.8 Specifying *onEntry* and *onExit* Handlers

In GSL *onEntry* and *onExit* handlers[8] are defined within the *state* clause as follows:

```
    state('state1') {
      oEntry namedClosure, {/*in-place closure*/}, //more closures
      //the rest of state specification goes here
    }

//somewhere in the controller
def namedClosure = {/*do something*/}
```

In GSL, all actions in transitions, *onEntry*, and *onExit* handlers are defined using Groovy closures[13]: tiny anonymous classes which have full access to the enclosing class (e.g., the controller) internals and encapsulating a particular functionality (see the EPG example above and Calculator example below).

4.9 Specifying Transitions

In GSL, transitions are specified using the so called *fluent* style widely used in many embedded DSLs[2]. In its full form, the transition specification looks as follows:

```
    state('state1') {
      on('trigger').guard(guardClosure)
                   .action(actionClosure).to('target1')
    }

//somewhere in the controller
def actionClosure = {/*do something*/}
def guardClosure  = {/*check something*/}
```

As with *onEntry* and *onExit* handlers, the transition specification could use named or in-place closure for actions and guards. Following the UML 2.0 specification[8], the main actions are supposed to perform some modifications in Model or View, while guards are supposed to validate some conditions (normally in Model).

4.10 Specifying Dynamic Choice and Action Pseudo-states

In GSL, dynamic choice pseudo-state[8] specification is similar to a normal state, but without *onEntry* and *onExit* handlers, nested states and *on()* trigger:

```
    choice('label') { //label is optional
      //conditional transition:
      guard(guardClosure).action{actionClosure}.to('target1')
```

```
      //default, else, transition:
  action(actionClosure).to('target2')
    }

//somewhere in the controller
def actionClosure = {/*do something*/}
def guardClosure  = {/*check something*/}
```

As with normal transitions, the *action* part is optional. If a dynamic choice pseudo-state contains only one default (else) transition, it could be specified in the form of an *action()* statement as follows:

```
    action('label') {
      action(actionClosure).to('target2')
    }

//somewhere in the controller
def actionClosure = {/*do something*/}
```

If a dynamic choice pseudo-state is used only for an initial pseudo-state, it could be specified without a label.

4.11 Advanced GSL Features

In the previous section, we described all basic GSL features: state machine, state, initial pseudo-state, onEntry, onExit, transitions, actions and guards. In order to illustrate more advanced features, we will use a more complex example - a basic calculator originally used in[9] (see Fig.6).

This statechart, though non-trivial but much simpler than its original version presented in [9], illustrates the point that statecharts, as a visual tool, still have a scalability challenge: almost any real-life application would lead to a diagram which would be hard to manage. Having an adequate and formal textual representation turns out to be a practical necessity and GSL was especially designed to address this need.

The corresponding Groovy script is presented below:

```
package calculator
import org.gsl.groovy.StateMachine
class UIController extends StateMachine {
  void define() {
    initial state('CALC') {
      on('C').to('CALC')

      initial state('FIRST_OPERAND') {
        onEntry eraseOperands
```

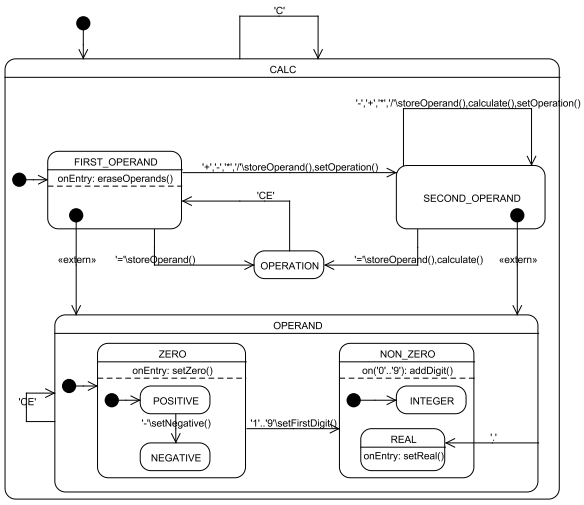

Fig. 6. Basic Calculator

```
  on('+','-','*','/')
    .action(storeOperand,setOperation).to('SECOND_OPERAND')
  on('=').action(storeOperand).to('OPERATION')
  initial state('OPERAND')
}

state('SECOND_OPERAND') {
  on('+','-','*','/')
    .action(storeOperand,calculate,setOperation).to('SECOND_OPERAND')
  on('=').action(storeOperand,calculate).to('OPERATION')
  initial state('OPERAND')
}

state('OPERATION') {
  on('+','-','*','/').action(setOperation).to('SECOND_OPERAND')
  on('CE').to('FIRST_OPERAND')
}

state('OPERAND') {
  on('CE').to('OPERAND')
  on('.').to('REAL')

  initial state('ZERO') {
    onEntry setZero
```

```
          on('1'..'9').action(setFirstDigit).to('NON_ZERO')

          initial state('POSITIVE') {
            on('-').action(setNegative).to('NEGATIVE')
          }
          state('NEGATIVE') {}
        }
        state('NON_ZERO') {
          on('0'..'9').action(addDigit)

          initial state('INTEGER') {}
          state('REAL') {
            onEntry setReal

            on('.') /*to disable the ancestor's handler */
          }
        }
      }
    }
  }

  def parts
  def operands
  def ops = [
     '+':{a1,a2->a1+a2},
     '-':{a1,a2->a1-a2},
     '*':{a1,a2->a1*a2},
     '/':{a1,a2->a1/a2}
  ]
  def operation
  def current

  String getResult() {
    parts[0] + parts[1] + '.' + parts[2]
  }
  def setZero = {
    parts   = ['','0','']
    current = 1
  }
  def setNegative = { parts[0] = '-' }
  def setFirstDigit = {String d ->
    parts[1] = d
  }
  def addDigit = { String d ->
    parts[current] += d
```

```
  }
  def setReal       = {current = 2}
  def eraseOperands = {operands = []}

  def storeOperand  = {
    operands << getResult().toBigDecimal()
  }
  def setOperation  = {String o-> operation = ops[o]}
  def calculate     = {
    def BigDecimal a2 = operands.pop()
    def BigDecimal a1 = operands.pop()
    def BigDecimal res= operation(a1,a2)
    operands << res
    parseResult(res)
  }

  private void parseResult(final BigDecimal res) {
    def String sRes = res.toString()
    parts = ['']
    if(sRes.startsWith('-')) {
      parts[0] = '-'
      sRes = sRes.substring(1)
    }
    def p = sRes.split('\\.')
    if(1==p.size())
      parts << sRes << ''
    else {
      parts << p[0] << p[1]
    }
  }
}
```

As with the previous example, the code has an almost one-to-one correspondence with the original statechart. The script demonstrates a number of advanced GSL features as follows:

1. Multiple triggers and Closure Arguments
2. Direct transition to inner states
3. **extern** States [2]

4.12 Specifying Multiple Triggers and Closure Arguments

If more than one event could trigger the same transitions than all triggers can be specified using a normal Groovy list and/or range notation as follows:

[2] **extern** states mechanism is a proprietary GSL extension intended to allow usage of common state specification similar to external procedures in conventional programming languages

```
    state('stateLabel') {
  //range specification:
  on(low..high).guard(guardClosure)
     .action(actionClosure).to('targetLabel')
  //list specification:
  on(t1,t2,t3).guard(guardClosure)
     .action(actionClosure).to('targetLabel')
  //mixed specification:
  on(low1..high1,t1,t2,low2..high2)
      .guard(guardClosure).action(actionClosure).to('targetLabel')

    }

//somewhere in the controller
def guardClosure  = {x -> /*check something using argument 'x'*/}
def actionClosure = {x -> /*do something using argument 'x'*/}
```

Multiple trigger specification is just a syntactic short cut and internally is translated into regular, single trigger, transaction. The main rationale behind this design decision was that multiple triggers would seldom constitute any serious scalability problem for run-time and therefore do not justify any complication of the simple trigger-to-transition mapping currently being used.

When multiple triggers are used, there is usually a need to pass the actual trigger as an argument to the corresponding guard and/or action closure. In GSL, this is accomplished through the regular Groovy syntax:

```
def guardClosure  = {ArgType arg -> /*check something*/}
def actionClosure = {arg -> /*do something*/}
```

Note 1. The argument type specification is optional.

4.13 Specifying Direct Transition to Inner States

Sometimes, as in the Calculator example above, there is a need to pass directly to some inner state. In the case of the Calculator, we would like to specify that whenever a '.' (dot) character is entered, we have to pass to the *real* state, which is a sub-state of the *non-zero* state. This in turn raises an important question of state labels name spacing. There are two basic options to consider:

1. Global namespace: all states need to have unique labels across the whole state machine
2. Scoped namespace: each super-state has its own name space

Supporting a direct visual representation of nested states was specified as an explicit design goal for GSL. The resulting notation looks very similar to nested classes or namespaces of the host programming language. Therefore, using scoped namespaces seems to be a natural choice. This, however, introduces a new

problem with target specification in transitions since following the strict scoping rule, we would need to put an outer state prefix for almost any transition. In other words, rather than specify it using more intuitive and short form:

```
state('state1') {
  on(trigger).to('state2')
  ...
}
state('state2') {
  ...
}
```

One would need to use a more verbose, although more formally correct, form:

```
state('state1') {
  on(trigger).to('outerState.state2')
  ...
}
state('state2') {
  ...
}
```

After spending some time on evaluating the both options we decided to adopt for GSL the global state namespacing schema since the scoped namespaceing, even though it is more formally correct, might introduce too much confusion.

Whether the global namespaceing schema is better than the scoped namespacing is too early to tell. Only after gaining some practical experience, a final decision regarding state labels namespacing could be made.

4.14 Using External States

As was clarified above **extern**, states do not belong to the UML 2.0 formal specification. However, they play a crucial role in eliminating duplications in statecharts. The statechart of the basic Calculator presented in [9] is significantly more complex than the sample we are using here. This additional complexity is caused by the fact that operand handling (from zero to non-zero, from positive to negative, from integer to real) needs to be duplicated for each operand, first and second. This duplication problem cannot be solved by using nested states only. Theoretically, it could be quite elegantly solved by using parallel states or regions (see Fig. 7).

Unfortunately, using parallel states just for eliminating duplications is not practical: many synchronization problems appear and even the basic processing algorithm needs to be complicated significantly. After spending some time evaluating the pros and cons of different possible solutions, we tend to agree with Miro Samek, who suggested in [9] to restrict parallelism to the state machine level only. At least for State-Oriented programming, this is a much more reliable and cost-effective solution and is fully in sync with the current trend to

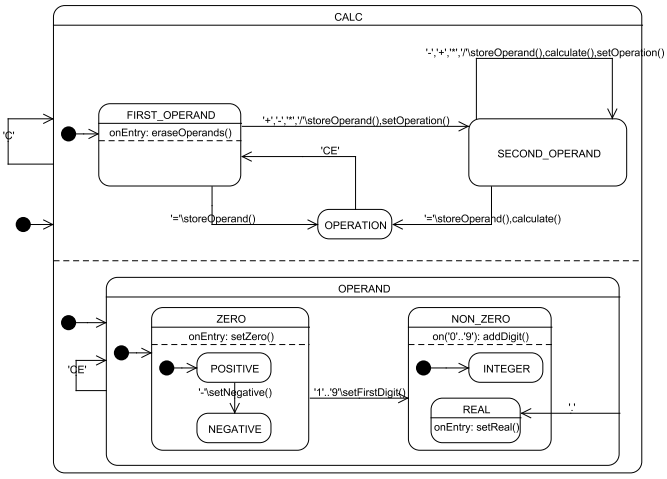

Fig. 7. Basic Calculator Statechart: Using Regions

implement concurrent processing systems by applying the *Active Object* design pattern[9, 5, 4].

In a sense the **extern** states solution is similar to using common subroutines in conventional procedural languages: a common behavior is extracted in a separated state, which can be *called* by a number of other states. In the case of the basic Calculator operand handling is common for the first and second operands and just duplicating its specification would be wasteful. Apparently, a very simple and cost-effective solution for **extern** states is possible. The current notation for **extern** states follows a minimalist approach:

```
state('stateA') {
  initial state('C') \\state A will use an extern state C
}
state('stateB') {
  initial state('C') \\state B will use an extern state C
}
state('stateC') { \\state C will be used by A and B
  ...
}
```

4.15 Other UML 2.0 Stetecharts Features

Some UML 2.0 statechart elements which have yet to be implemented by GSL are planned for the next version:

1. Final Pseudo-state

2. Deep and Shallow History Pseudo-states
3. External Transitions
4. Static Choice
5. Fork and Join Pseudo-states
6. Regions

We do not anticipate any serious challenge with the implementation of Final and History Pseudo-states as well as External Transitions: they are all expected to lead to a straightforward implementation which fully conforms to the already established design structure of GSL.

Implementing Static Choice[8] could be slightly more challenging due to a certain tension between notational clarity and run-time efficiency.

Supporting parallel states through regions and fork and join pseudo-states will open a whole new chapter. As stated above, using parallel states for just eliminating duplication is apparently impractical and the **extern** states solution is proposed as an alternative. Still, parallel processing is too important to be ignored, especially considering the current multi-core CPU trend[4]. As stated above, we suggest limiting the usage of parallel states including fork and join pseudo-states to independent state machines. In order to avoid synchronization problems, the most likely approach would be to apply a copy-on-write strategy for Model and (m.b. View) data. Additional research into alternative solutions for smooth host language integration is required. In this regard, Scala agents[15] seem the most promising direction.

5 Concluding Remarks

The State-Oriented Programming paradigm provides a powerful complementary tool for effective development of even-driven systems all the way from GUI applications, communication protocols to enterprise integration and Service Oriented Architecture[19]. The Generic Statechart Library (GSL) presented in this paper suggest an easy-to-use and economical way of embedding an essential subset of UML 2.0 statechart formalism directly in modern Object-Oriented scripting languages such as Groovy, Ruby, Scala, etc.

GSL follows a minimalist run-time model in many aspects inspired by C++[20]: no distributed fat, simple, preferably fixed-time algorithms, and small foot print data structures. Even in its current form, GSL is suitable for a wide range of applications, including rich client applications, Web applications, SOA services, testing tools and simulators. A natural next step would be the integration with popular Java GUI packages such as Java Swing, JavaFX and maybe Adobe Flex, as well as with Web application frameworks such as Grails (Groovy), Ruby on Rails, and lift (Scala).

GSL syntax was carefully chosen to stay as close as possible to the host programming language on the one hand and to the UML 2.0 statechart formalism on the other hand. This should, in principle, facilitate a smooth integration with IDE tools such as Eclipse in order to support advanced debugging, code

completion, syntax coloring and statechart refactoring. The latter is a very interesting topic by itself. These productivity tools would allow supporting the State-Oriented Programming as a first class citizen paradigm. Once they are in place, only the sky is the limit.

References

1. M. Fowler, Patterns of Enterprise Application Architecture, Adisson-Wesley, 2002
2. M. Fowler, Domain Specific Languages book, work in progress, available at http://www.martinfowler.com/dslwip/
3. J. Huges, Why Functional Programming Matters, Chalmers memo, 1984, available at http://http://www.cs.chalmers.se/ rjmh/Papers/whyfp.html
4. J. Armstrong, Programming Erlang: Software for a Concurrent World, Pragmatic Bookshelf, 2007
5. F. Buschmann et al, Pattern-Oriented Software Architecture Volume 4: A Pattern Language for Distributed Computing, Wiley, 2007
6. T. Parr, The Definitive ANTLR Reference: Building Domain-Specific Languages, Pragmatic Bookshelf, 2007
7. D. Harel, Statecharts: A Visual Formalism for Complex Systems, Sci. Comput. Program, 1987, vol. 8, 231-274
8. OMG Unified Modleling Language (OMG UML), Superstructure: V2.1.1, 2007
9. Miro Samek, Practical Statecharts in C/C++: Quantum Programming for Embedded Systems, CMP Books, 2002
10. The Boost Statecharts Library, available at http://www.boost.org
11. State Chart XML (SCXML): State Machine Notation for Control Abstraction 1.0, available at http://www.w3.org/TR/2005/WD-scxml-20050705/
12. Apache Commons SCXML, available at http://commons.apache.org/scxml/
13. D.Koening, et al, Groovy in Action, Manning Publications, 2007
14. D.Thomas, et al, Programming Ruby: The Pragmatic Programmers' Guide, Second Edition, Pragmatic Bookshelf, 2007
15. M. Odersky, et al, Programming in Scala, Aritma Developer, 2008
16. E. Gamma, et al, Design Patterns: Elements of Reusable Object-Oriented Software , Addison-Wesley, 1994
17. M. Cohn, User Stories Applied: For Agile Software Development, Adisson-Wesly, 2004
18. R. Mugridge, W. Cunningham, Fit for Developing Software: Framework for Integrated Tests, Prentice Hall PTR, 2005
19. G. Hohpe, B. Woolf, Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions, Addison-Wesley , 2003
20. B. Stroustrup, The Design and Evolution of C++, Addison-Wesley, 1994