ELEKTROTEHNIŠKI VESTNIK 78(4): 223–228, 2011
ENGLISH EDITION

SimpleFSM - a domain-specific language for SIP communication systems - Part I: Language description

Edin Pjanić, Amer Hasanović

Faculty of Electrical Engineering, University of Tuzla,
Franjevačka 2, Tuzla 75000, Bosnia and Herzegovina
E-mail: {edin.pjanic, amer.hasanovic}@untz.ba

Abstract. This two-part paper demonstrates an application of metaprogramming techniques to the development of domain specific languages (DSL) using the Ruby programming language and its application to SIP communication systems. Part I proposes the SimpleFSM, a DSL for finite-state machines (FSM), together with simple application examples. Part II proposes an approach to the SIP application development using the developed DSL in order to speed up and simplify the development process.

Keywords: Domain-specific languages, Session-Initiation Protocol, finite-state machine, metaprogramming, Ruby, telecom applications.

1 INTRODUCTION

Communication applications involving different devices (computers, personal digital assistants, mobile phones) and exchanging voice, video and data are becoming increasingly popular. Skype, Google Talk, Google Wave and Microsoft Live are all examples of such services. These services utilize various communication protocols, either standard or proprietary. The most widely used communication protocol, especially in the telecom industry, is the Session Initiation Protocol (SIP) protocol.

SIP is a text protocol standardized in RFC 3261 [1] with a syntax similar to the Hypertext Transfer Protocol (HTTP) developed and used for establishing and managing multimedia IP sessions. To support the SIP application development, Java Community Process released the SIP Servlet API specified in JSR116 [2] and JSR289 [3]. The SIP servlets are typically deployed to Java enterprise application servers (JEE), such as SailFin [4] or Mobicents [5].

The SIP Servlet API specification is modeled to mimic the HTTP Servlet API. This offers an easy transition to the SIP application development for a large community of web application developers already familiar with the HTTP servlet model. However, HTTP and SIP protocols are different in essence. While HTTP is a stateless protocol based on the request/response model, SIP is a stateful protocol that can involve multiple call flows between different call parties originating from SIP and web clients. Utilizing only the SIP Servlet API to handle such call flows could become cumbersome.

In order to speed up and simplify the SIP application development, various approaches have been proposed. One such approach is based on combining the SIP Servlet API and ECharts [6], an open-source domain-specific language (DSL) [7] for modeling finite-state machines (FSM).

ECharts uses a textual syntax, supports hierarchical concurrent state machines, machine reuse and multiple transition priority levels. ECharts is a hosted language, which means that ECharts relies on the host language statements embedded in the machine definition. The ECharts code therefore has to be translated to Java in order to be compiled and deployed to a Java EE server. ECharts solves many of the problems related to handling of complex SIP state machines. However, the fact that it has to be compiled twice, first to get the corresponding Java code then to get the bytecode, makes it difficult to use it within short and rapid development cycles found in modern software development practices. Furthermore, a new language has to be adopted in order to develop SIP applications based on this programming model.

The web development community has recently adopted several DSLs based on dynamic programming languages. Examples of such DSLs are Ruby on Rails [8] based on Ruby, and Django [9] based on Python. These DSLs have been proven effective in short and rapid application development cycles [10]. In order to speed up and simplify the development process of telecom applications, it would be beneficial to design a DSL for SIP communication systems based on a dynamic programming language. The concepts of this aproach were proposed in [11].

This two-part paper proposes a domain specific language for FSM using the Ruby programming language and its application to SIP communication systems. The

Received October 5, 2011
Accepted October 24, 2011

developed DSL, called SimpleFSM, is implemented as an internal DSL using an embedded implementation pattern [12], [13], [14] and is primarily designed for SIP communication systems, but can be used for modeling an FSM for any domain. It can be combined with SIP servlets to produce SIP applications entirely in the Ruby programming language.

Part I of this two-part paper focuses on the development techniques used to design and implement the SimpleFSM DSL. Section 2 gives a brief overview of some important features found in the Ruby programming language and its Java implementation, i.e. JRuby. The most important metaprogramming features of Ruby are discussed in Section 3. Section 4 describes SimpleFSM, a simple but complete Ruby DSL for FSM and gives some application examples. The techniques used to implement the FSM DSL are described in Section 5.

Part II [15] presents the application of the SimpleFSM DSL to SIP communication systems by combining it with SIP servlets. By utilyzing the SimpleFSM in our demo application we show that this DSL can be effectively used to simplify development of a SIP application with complex call flows.

2 RUBY AND JRUBY

Ruby is a dynamic language usually grouped with scripting languages, such as Smalltalk and Python. It has some powerful features, missing in C++ and Java, such as:

- full object orientation, since everything in Ruby is an object,
- dynamic typing, dynamic classes and objects that can change during runtime,
- native support for regular expressions and containers, and
- support for metaprogramming in order to develop DSLs.

The reference implementation of Ruby, written by Yukihiro Matz Matsumoto, is an open-source software developed in C and is currently in version 1.9. Another open-source implementation is called JRuby. It is fully compatible with the reference implementation and implemented in Java on top of Java Virtual Machine (JVM). Compared to the C implementation of Ruby, JRuby has the following advantages:

- better performance on standard Ruby benchmarks,
- kernel threading support via Java threads,
- native Unicode support,
- integration with Java libraries, since any Java class inside JRuby is a valid Ruby class, and
- easier path to getting Ruby in the enterprise.

Furthermore, JRuby fully supports the Ruby's metaprogramming model. Hence, the existing DSLs developed with that programming model, such as Ruby on Rails and Sinatra web DSLs, Active Record and DataMapper database DSLs and others, are automatically available in JRuby.

3 DSL SUPPORT IN RUBY

Ruby is a dynamically typed programming language [16]. In the context of creating DSLs, this feature is very important considering that classes and objects can be dynamically created and modified during runtime with custom methods and members depending on the context of invocation. Ruby methods can accept and return any number and type of arguments.

Furthermore, Ruby supports sending code blocks to methods as arguments. A block is similar to an anonymous function or a chunk of code. The method can execute the received block of code when required, but in the context where the block was created. This is often referred to as a closure. Ruby also supports procs and lambdas, which are similar to code blocks, but can be manipulated as objects.

Finally, special methods, such as `eval`, `class_eval`, `instance_eval`, `send` etc., are very important for metaprogramming support in Ruby. The following section gives an overview of how these concepts can be utilized to develop a DSL for FSM in Ruby.

4 SIMPLEFSM - A DSL FOR MODELLING FINITE-STATE MACHINES

Traditionally, in programming languages, such as C, C++ or Java, FSMs are implemented using switch statements or complex object structures. Metaprogramming techniques can be utilized to develop a descriptive, powerful, clear and simple DSL to support FSM programming concepts. In this section one such DSL is described. The entire implementation is less than 300 lines of code. However, only the most important parts are analyzed in this section.

The developed DSL can be used to model an FSM for any domain, including complex communication applications based on the SIP protocol. The DSL was developed in order to support the following requirements:

- unlimited number of states can be used,
- unlimited number of transitions can be specified,
- state transition can be conditional,
- an action can be invoked on entering a state and/or exiting a state,
- an action can be executed on an event, and
- events can receive an arbitrary number of arguments which are sent to all related actions during the event processing.

FSM actions are modeled using Ruby methods. This makes the FSM model compact and clear. The en-

tire DSL is implemented in a Ruby module called `SimpleFSM`.

To utilize the DSL in a new class, the DSL module should be included into the class. The state machine, that is implemented in the class, is defined within the block of code after the `fsm` keyword, as shown in the example code given in Listing 1.

Listing 1 Example of a `Worker` FSM model

```
class Worker
  include SimpleFSM
  fsm do
    state :resting
    state :working,
      :enter => :ckeck_in,
      :exit => :check_out
    transitions_for :resting do
      event :work, :new => :working
    end
    transitions_for :working do
      event :rest, :new => :resting
    end
  end

  private
  def check_in(args)
    #- further code omitted -#
  end

  def check_out(args)
    #-further code omitted-#
  end
end
```

Listing 1 defines a class named `Worker` whose state diagram is depicted in Fig. 1. Inside the block named `fsm` a custom language is used to define the FSM. Two states, `:resting` and `:working`, are defined using the `state` statement. The statement accepts optional parameters, `:enter` and `:exit`. These parameters can be used to specify actions that are executed when entering or leaving the state that is being defined. Hence, for the `Worker` class, methods `check_in` and `check_out` are executed respectively, every time the state `:working` is entered or left.

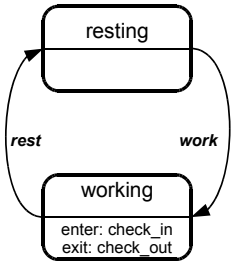

Figure 1. Worker state diagram

FSM state transitions are defined within the `transitions_for` statement. The arbitrary number of transitions for any state can be specified using the `event` statement. For the `Worker` class, the two transitions were defined. When the class is in the `:resting` state, event `:work` triggers a transition to the `:working` state. Similarly, event `:rest` fires a transition from `:working` to the `:resting` state. The FSM remains idle when an event is received which is not specified in the `transition_for` statement related to the current state.

The following is the full list of parameters that the `event` specification accepts inside the `transition_for` statement:

- `:new` specifies the destination state for the transition. The parameter is mandatory. If `:new` is `nil`, event is triggered but the transition is not performed and the FSM remains in the same state.
- `:guard` specifies the Boolean function for checking the transition's condition. The parameter is optional. The event is triggered and transition is performed only if this method returns `true`.
- `:do` specifies the method to be called when the event is fired. This parameter is optional. If `:guard` is specified, then the `:do` method is called only if the `:guard` method returns `true`.

Since Ruby programmers are acustomed to different styles of code writing, two aditional versions of `transitions_for` specification are supported, both without using the `do-end` block. The transitions from Listing 1 can be written in the following forms:

```
transitions_for :resting,
    event(:work, :new => :working)
transitions_for :working,
    event(:rest, :new => :resting)
```

or

```
transitions_for :resting,
  {event => :work, :new => :working}
transitions_for :working,
  {event => :rest, :new => :resting}
```

States specified as `:new` in any `transitions_for` statement are created if they are not explicitly defined using the `state` statement. However, if `:exit` and `:enter` actions for the state are required the `state` statement must be used. States specified in the `fsm` block will become available in the objects that are instances of the class with the `fsm` specification. For every event in the `fsm` block, an object will get a method with the same name, which can be used to generate the event. The following code is used to demonstrate these features.

```
foo=Worker.new
foo.run
foo.work
foo.rest
```

After the object `foo` is created, the transition to the initial state is triggered by invoking the `run` method. The initial state is the first state defined in the `fsm` block. After that, the FSM within the object is ready to accept events. By calling the methods `work` and `rest` on the object `foo`, events `:work` and `:rest` are generated and the incorporated FSM is manipulated accordingly.

4.1 Example - Vending machine

In order to demonstrate the more advanced features of the DSL, a vending machine model is used. The vending machine accepts coins and serves several types of beverages. For any beverage, a price can be set. When a user inserts a coin into the machine, the information about the credit amount and available beverages is shown. The user can select a beverage based on the inserted amount of money, or press the cancel button to retrieve the remaining credit.

The described behavior can be modeled by an FSM with three states and four events, which is shown in Fig. 2.

The state diagram from Fig. 2 is implemented in the `Vending` class. The code segment of this class that implements the FSM is shown in the Fig. 2 listing and two selected private methods of the class that are invoked during FSM transitions are shown in Listing 2. Beverage prices and valid coins are specified in instance variables `@prices` and `@valid_coins`.

The Vending class can be utilized in any scenario a typical vending machine would be used. The following code demonstrates one such use case:

```
machine = Vending.new
machine.run
machine.coin 1
machine.coin 'strange coin'
machine.coin 1, 2, 0.5
machine.select :cappuccino
machine.cancel
```

In the above code, a vending machine object is instantiated, started and finally, the five events are generated in a sequence.

Events defined in the `fsm` block can receive an arbitrary number of arguments. In this example, the `coin` event method receives the inserted coin values and the `select` event method receives a parameter that specifies the selected beverage type. The same arguments used to call these methods are passed to the related `:guard`, `:do`, `:exit` and `:enter` methods, as specified in `transition_for` statements. These arguments are always packed in a single array.

The first event in the use case is invoked by calling the `coin` method of the `machine` object with 1 as an argument. The FSM facility evaluates transitions in lines 15 and 16 of the Fig. 2 listing. The `:guard` method of the first transition, `coin_valid?`, is called with a single element array as an argument. As shown in Listing 2, for every element of `args` array, `coin_valid?` method checks if it matches any element of the `@valid_coins` array. The value `true` is returned only if all elements are matched. Otherwise, `false` is returned. The `:guard` method of the second transition, `coin_not_valid?`, checks if there are invalid coins in the argument array. Since 1 is evaluated as a valid coin by the `:guard` method `coin_valid?`, the transition condition specified in line 15 of the Fig. 2 listing is satisfied and this transition is performed. The `:do` method `update_credit` associated with this transition is also executed with the same argument. For each element of the received `args` array, method `update_credit` increments the credit amount if the element is either of a `Fixnum` or a `Float` type, as shown in Listing 2. After that, the actual transition to the state `:credit` is performed. Upon entering the state `:credit`, the `msg_menu` `:enter` method is executed as defined in the `:credit` state definition.

The second event in the sequence will trigger the transition specified in line 23, because the inserted `'strange coin'` is not a valid coin according to the `coin_not_valid?` guard condition. In this case the FSM remains in the same state without invoking `:enter` and `:exit` methods.

The third `coin` event is invoked with three values. This simulates successive insertion of three coins. Since the coins are valid, the appropriate transition in this case is the one specified in line 22 of the Fig. 2 listing. The method `update_credit` is called again and the credit is incremented to 4.5 . This time the FSM reenters the same state and executes the specified `:exit` and `:enter` methods.

Next, invoking the `select` event method, with `:cappuccino` as an argument, triggers the transition in line 25. During the transition to the destination state `:serving`, the `:do` method `prepare_drink` is invoked. The `:enter` method of the `:serving` state, `serve_drink`, decrements the credit amount for the beverage price by calling the `update_credit` method, prints a message and generates the `:served` event by calling the `served` method, as shown in Listing 2. After that, the new transition to the state `:credit` is performed.

Finally, the `cancel` event is invoked, the `:do` method `return_credit` is called and the FSM tran-

Listing 2 Selected private methods of the `Vending` class

```
def update_credit args
  if args
    args.each do |v|
      if v.is_a?(Fixnum) or v.is_a?(Float)
        @credit += v
      end
    end
  end
  msg_credit args
end

def coin_valid? args
  if args
    args.all? do |coin|
      @valid_coins.any?{|v| v == coin}
    end
  else
    false
  end
end
```

```ruby
class Vending
  include SimpleFSM
  def initialize
    @credit ||= 0
    @prices = { :coffee => 1, :tea => 1, :cappuccino => 2}
    @valid_coins = [0.5, 1, 2, 5]
  end

  fsm do
    state :idle
    state :credit, :enter => :msg_menu
    state :serving, :enter => :serve_drink

    transitions_for :idle do
      event :coin, :new => :credit, :guard => :coin_valid?, :do => :update_credit
      event :coin, :new => nil, :guard => :coin_not_valid?, :do => :return_coin
      event :select, :new => nil, :do => :msg_not_enough
    end

    transitions_for :credit do
      event :cancel, :new => :idle, :do => :return_credit
      event :coin, :new => :credit, :guard => :coin_valid?, :do => :update_credit
      event :coin, :new => nil, :guard => :coin_not_valid?, :do => :return_coin
      event :select, :new => nil, :guard => :not_enough_credit?, :do => :msg_not_enough
      event :select, :new => :serving, :guard => :enough_credit?, :do => :prepare_drink
    end

    transitions_for :serving do
      event :served, :new => :credit, :guard => :has_credit?
      event :served, :new => :idle, :guard => :no_credit?
    end

  end

  private
  #-further code omitted-#
end
```

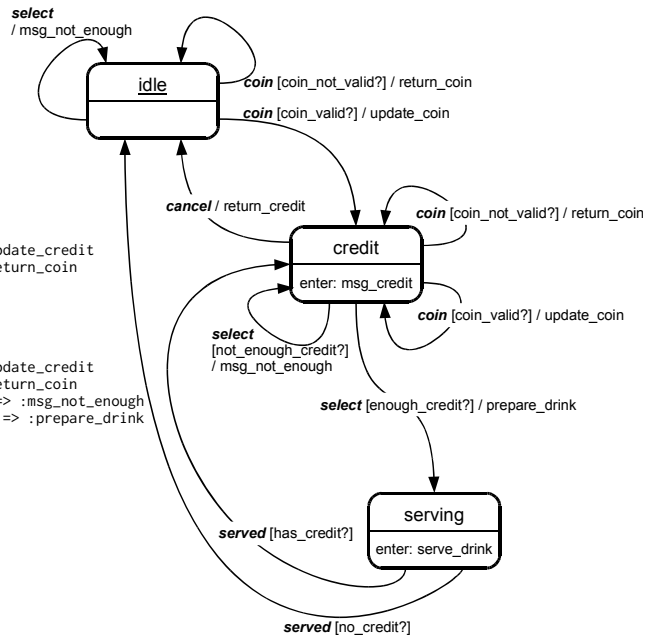

Figure 2. The vending machine class implemented using the developed DSL and its state diagram

sitions to the `:idle` state.

In this section we have presented the basic principles of FSM modeling and implementation in Ruby using the SimpleFSM DSL. In Part II of this two part paper, we apply the described principles to SIP communication system and give a more complex example, where we combine SimpleFSM DSL with Java SIP Servlets, in order to develop a practical click to call application.

5 DSL IMPLEMENTATION DETAILS

The developed DSL uses the pure Ruby syntax. Furthermore, it does not require compilation or specialized parsers. When a class includes the `SimpleFSM` module, Ruby injects into the class: class variables, instance methods and class methods that are used to model the FSM. The injected class methods can be called against the class, while the instance methods can be invoked on class objects. The most important injected class method is the `fsm` method. This method is usually invoked inside the targeting class after inclusion of the `SimpleFSM` module. The `fsm` method expects to receive a code block which is used to describe the FSM. The statements of the FSM language inside the `fsm` code block, such as: `state`, `transitions_for` and `event`, in reality are class methods which, when invoked, manipulate the injected class variables of the targeting class.

When a class includes the `SimpleFSM` module that is specified in Listing 3, the `included` method of the module is executed with the target class being passed as the `klass` parameter. By invoking the `class_eval` method on the `klass` object, two important tasks are performed. First, the variables: `@@states`, `@@events` and `@@transitions`, used to store the model data about the states, events and transitions, are injected as class variables into the class. Second, the meth-

Listing 3 Part of code of the `SimpleFSM` module

```ruby
module SimpleFSM
  def self.included klass
    klass.class_eval do
      @@states ||= []
      @@events ||= []
      @@transitions ||= {}

      def self.fsm (&block)
        instance_eval(&block)

        # Event-methods definition
        @@events.each do |ev|
          Kernel.send :define_method,
            ev do |*args|
              #-further code omitted-#
          end
        end
      end

      def self.state(sname, *data)
        #-further code omitted-#
      end
      def self.transitions_for(sname, *trans)
        #-further code omitted-#
      end
      def self.event(ev, args)
        #-further code omitted-#
      end
      #-further code omitted-#
    end
  end
  #-further code omitted-#
end
```

ods: `fsm`, `state`, `transition_for` and `event` are constructed and injected into the target class as class methods. After this operation is performed, the class recognizes the keywords of the FSM meta-language, most importantly the `fsm` keyword.

When the `fsm` keyword is encountered inside the class, the `fsm` class method gets invoked. The method receives a code block that describes the desired FSM behavior. Listing 3 shows that the `fsm` method first passes the received code block to `instance_eval`, which then executes the received block within the context of the class. When this happens, the class methods, that were previously injected during the `SimpleFSM` module inclusion, are invoked in order to setup the injected class variables according to the model definition. Finally, by utilizing the `Kernel.send` method, one instance method is dynamicaly created for every event definition in the model. Event names were previously stored in the `@@events` class variable of the `Array` type. The constructed event methods can receive arbitrary number of arguments that are mapped to the `args Array` parameter inside the event method. Event methods hold the required logic, so that when invoked on an instance of the class, they check and perform transitions according to the statements specified in the `transitions_for` definitions. There are other supporting class methods and instance methods that are also injected into the class, but are not shown in Listing 3 for simplicity.

6 CONCLUSION

This paper demonstrates a metaprogramming approach for domain specific language development within the Ruby programming language and its application to SIP communication systems.

The described approach is used to develop a DSL for FSM, which is convenient for modeling FSMs for any domain, including complex communication applications based on the SIP protocol.

The DSL has simple syntax for modeling FSMs. The FSM is modeled by defining states and transitions that can be conditional and can be performed on certain events. The DSL supports definition of actions that can be executed during transitions or on entering and exiting states. These actions are modeled as methods inside the class the FSM model is embedded in.

The DSL has a pure Ruby syntax and does not require parser or other facility in order to be used in Ruby applications. Moreover, the DSL is developed as an internal DSL, which means that it becomes an integral part of the language.

In Part II of this two-part paper, we demonstrate an approach to development of SIP applications using the SimpleFSM DSL.

REFERENCES

[1] J. Rosenberg, H. Schulzrinne, G. Camarillo, A. Johnston, J. Peterson, R. Sparks, M. Handley and E. Schooler. SIP: Session Initiation Protocol, RFC 3261 (Proposed Standard), IETF, updated by RFCs 3265, 3853, 4320, 4916, 5393, 5621
[2] A. Kristensen. SIP Servlet API, JSR 116
[3] Y. Cosmadopoulos and M. Kulkarni. SIP Servlet v1.1, JSR 289
[4] SailFin Project Website, https://sailfin.dev.java.net (1.9.2011)
[5] Mobicents Website, http://www.mobicents.org (1.9.2011)
[6] T. M. Smith and G. W. Bond. ECharts for SIP Servlets: a statemachine programming environment for VoIP applications, In: *Proc. IPTComm International Conference on Principles, Systems and Applications of IP Telecomunications (IPTComm 07)*, pp. 89-98, 2007.
[7] A. van Deursen, P. Klint, and J. Visser. Domain-specific languages: an annotated bibliography, *ACM SIGPLAN Notices*, Vol. 35, pp. 26-36, 2000.
[8] M. Bachle and P. Kirchberg, Ruby on Rails, *IEEE Software*, Vol. 24, pp. 105-108, 2007.
[9] The Django framework Website, http://www.djangoproject.com (1.9.2011)
[10] V. Viswanathan. Rapid Web Application Development: A Ruby on Rails Tutorial, *IEEE Software*, Vol. 25, no. 6, pp. 98-106, 2008.
[11] E. Pjanić, A. Hasanović, N. Suljanović, A. Mujčić and M. Zajc. Metaprogramming approaches to finite state machine modeling for SIP applications, In: *Proc. MELECON 2010 - 15th IEEE Mediterranean Electrotechnical Conference*, pp. 592-596, 2010.
[12] M. Mernik, J. Heering and A. M. Sloane. When and how to develop domain-specific languages, *ACM Comput. Surv.*, Vol. 37, no. 4, pp. 316-344, 2005.
[13] J. Cuadrado and J. Molina. A model-based approach to families of embedded domain-specific languages, *IEEE Transactions on Software Engineering*, Vol. 35, no. 6, pp. 825-840, 2009.
[14] S. Gunther, M. Haupt and M. Splieth. Agile engineering of internal domain-specific languages with dynamic programming languages, In: *Proc. 2010 Fifth International Conference on Software Engineering Advances (ICSEA)*, pp. 162 -168, 2010.
[15] E. Pjanić and Hasanović. SimpleFSM - a domain specific language for SIP communication systems - Part II: Application to SIP Servlets, *Elektrotehnički vestnik*, (submited for publication).
[16] L. Tratt. Dynamically Typed Languages, *Advances in Computers*, Vol. 77, pp. 149-184., 2009.

Edin Pjanić received the M.Sc. degree from the University of Tuzla, Bosnia and Herzegovina, in 2005. He is currently working toward the Ph.D. degree at the same university. He is a teaching assistant at the University of Tuzla. His research interests include rapid web and telecom applications development, dynamic programming languages and domain specific languages.

Amer Hasanović was born in Bosnia and Herzegovina. He received the Diplomirani Inženjer Elektrotehnike degree from the University of Tuzla, Bosnia and Herzegovina, in 1999, and MS (2001) and PhD (2004) from West Virginia University, Morgantown, USA. He joined faculty of Electrical Engineering, University of Tuzla in September 2004 where he is now an Associate Professor. He has been working in the field of robust decentralized control and component oriented software design for large scale systems simulations. His current research interest is in the field of telecom and web applications development based on dynamic programming languages.