Creation of Automaton Classes from Graphical Models and Automatic Solution for Inverse Problem

Yuri Gubin
DataArt
ygubin@dataart.com

Kirill Timofeev
DataArt
ktimofeev@dataart.com

Anatoly Shalyto
SPbSU ITMO

Abstract

Graphical models – integral part of any program development. Using graphical models facilitates analysis of their architecture and understanding of logic.

This work shows how to automatically transfer automaton code into graphical isomorphic model, using Java programming and DOT modeling languages.

1. Introduction

Currently in the design of programs with complex behavior actively used tools for graphical representation of logic and structure, for example, UML[1]. They make it possible to analyze the program, working with its components, rather than source code. The graphical presentation makes it easy to debug, and provides the means to address possible shortcomings and mistakes.

Automaton library for graphical representation of state machine will be developed in this work using language with static types Java. It should to provide the possibility of creating automaton classes and automatically transfer them to isomorphic graphic model. This library implements the domain-specific language (DSL, Domain Specific Language [2]), which will allow:

- to check and modify the program without any knowledge of Java programming language by experts of the subject area;
- to provide failure–resistance of developed program;
- to work with the code using terms of subject area - to improve the readability of source code.

The library has been developing for object-oriented programs with explicit allocation of states and state machine to describe behavior of them [3].

The purpose of this work – the implementation of the automatic transfer of automaton classes (automated classes [3]) executable code to isomorphic graphical model (reverse engineering). As an example, automaton of user registration on Web site from the Restful-authentication plug-in [4] will be implemented.

2. Description of the algorithm of automatic transfer of the executable code to isomorphic graphical model

The proposed algorithm converting the following components of a state machine to a graphical representation:

1) states that do not belongs to any group;
2) groups, and nested groups;
3) nested states and transitions.

Components of state machine can be read using meta information of automaton class, or easily from arrays of elements.

The result of the algorithm will be a model in text language DOT [5]. This language provides ability to describe different graphical components (applied to automaton – states, transitions and groups). Also it allows editing styles of components.

As an example, a model of two parallel processes described in this language:

```
digraph G {// Name of oriented graph
 subgraph cluster0 {// process #1
  // Styles
  node [style=filled,color=white];
  style=filled;
  color=lightgrey;
  // Nodes and
  a0 -> a1 -> a2 -> a3; arcs
  // Name of the group
  label = "process #1";
 }
 subgraph cluster1 {// process #2
  node [style=filled];  // Style
  // Nodes and arcs
  b0 -> b1 -> b2 -> b3;
  // Name of the group
  label = "process #2";
  color=blue   // Border color
 }
 // Nodes and arcs
 start -> a0;
 start -> b0;
 a1 -> b3;
 b2 -> a3;
 a3 -> a0;
 a3 -> end;
 b3 -> end;
 // Styles for the initial and final
 // nodes
```

```
start [shape=Mdiamond];
end [shape=Msquare];}
```

Dot (dot) utility from graphviz [6] package is using to get the image from the DOT model description.

Figure 1 shows a graphical representation of the model obtained from the description in DOT language shown above.

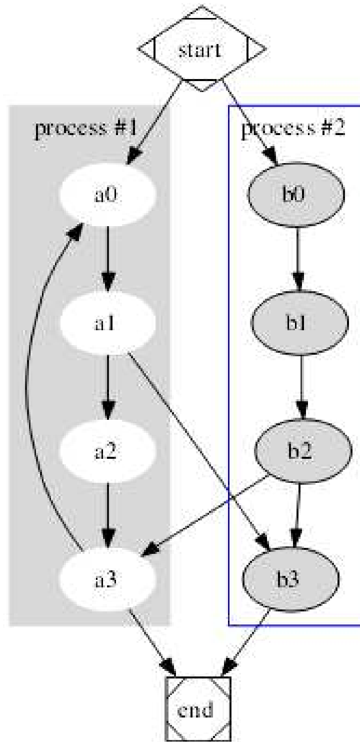

Figure 1. An example image described using the DOT language

It is known how to transform each component of automaton class into DOT language. States, groups and transitions can be represented as a formatted string. So, we need to process states firstly, then groups and nested states and later transitions. Using this order we would not create duplicated nodes in DOT model and result will be easy to use.

For example transition could be transformed into `start -> a0`; where `start` and `a0` is a names of states. State could be saved just as `a0` and group could be saved as

```
subgraph clusterGroup{
label="name of group"
;}
```

Getting presentation in DOT language from all components we can create an output model.

3. The implementation of the library to create an object-automaton programs

To ensure compliance with the objectives develop the automaton library, which will allow effectively implementing of the automaton classes firstly, and, secondly, will contain a class that provides automatic transfer of executable code to the graphical isomorphic model. The development of automaton library is taken into feature of further conversion of automaton Java classes into DOT text description.

The developed library includes the following classes:

- State – class, provides ability to specify functions at the entrance to the state and at the exit from it;
- Transition – transition between states and groups. Allows to specify functions during transition;
- StateGroup – group of states.
- DSL – base class for automaton classes. This class contains all needed common functionality for automaton class (i.e. methods for event reading, default constructors and etc.).The DSL class also includes method Compile for setting up initial state of automaton using metadata of automaton class.

Classes State, Transition and StateGroup inherited from common class Entity. This class provides basic properties, which allow uniformly processing of all components.

This library published in «Projects» section on http://is.ifmo.ru.

Creation of automaton classes for application is based on the library classes. Each of such automaton classes includes described method and all needed common functionality.

Feature of the developed library is that it uses anonymous classes for implementing functions in states and transitions. Anonymous class – is a local class without name. It has been creating and initializing in single expression [7]. This allow us to use them instead of lambda functions of Ruby language [4] for creating states and transitions with all needed functions without separated declaration of methods. Interfaces Guard and Action have been using to create described anonymous classes.

As an example, consider the graphical model of an automaton (Figure 2).

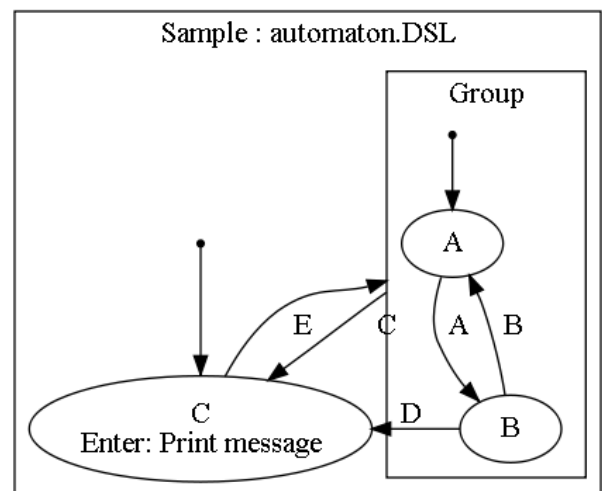

Figure. 2. Model of an automaton.

Create Java automaton class for proposed model with developed library.

```
import java.util.*;
import automaton.*;
public class Sample extends DSL{
  public State c = new State("C",
// Anonymous class for event on state
// entrance
     new Action() {
       public void go(){
         System.out.println(
           "Come to state C");}
       public String toDOT() {
         return "Print message";}
     },
     null,null); // No other events
  public StateGroup group =
    new StateGroup("Group");
  public Transition tgc =
    new Transition("GC","C",group,c);
  public Transition tcg =
    new Transition("CG","E",c,group);
// Will be created in constructor
  public Transition tbc;

// Constructor
  public Sample () {
    Vector <Entity> groupEntity =
      new Vector<Entity>();
    State a = new State("A");
    State b = new State("B");
    groupEntity.add(a);
    groupEntity.add(b);
    groupEntity.add(
      new Transition("AB","A",a,b));
    groupEntity.add(
      new Transition("BA","B",b,a));
     tbc = new
Transition("BC","D",b,c);
     group.setAll(groupEntity);
  }
}
```

Note that separated method for state entrance is not required in this code, because code was developed using anonymous class with implemented needed logic. Using of developed library provides:

- creation of methods for states and transitions by means of anonymous classes without duplicated separated methods [8];
- inheritance of automaton classes without additional tools;
- syntactic attractiveness.

4. Implementation of automatic isomorphic transfer algorithm in the library

In previous section it was shown how to create an automaton class with developed library. Proceed to consider the inverse problem.

Consider this algorithm on an example of automaton of user registration on Web site from the Restful-authentication plug-in. User registration consists from follow actions:

- fill the registration form;
- enter an activation code;
- fill the personal information.

Administrator can perform follow action:

- delete user and it personal information from system.

Figure 3 shows the expanded graphical notation [9], which describe automaton for registration.

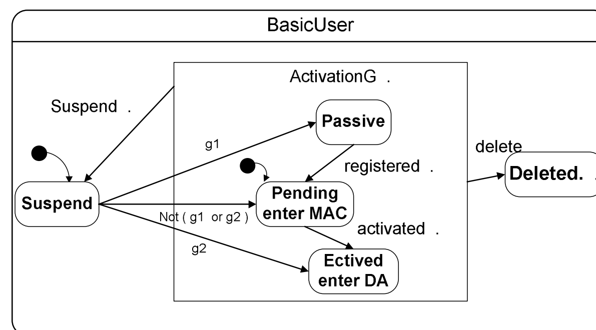

Figure 3. Automaton for registration

This automaton includes the following states:

- Suspended – user is waiting;
- Pending – user should to confirm registration by activation code;
- Active – user registered;
- Passive – user only logged in registration system;
- Deleted – user removed.

Group «ActivationG» contains states, in which user acts with registration system.

For its creation, it is necessary to define the collection of states and transitions for group. Determination of the group's structure occurs when the defined collection is passed to the constructor of group. Creation of group placed in constructor of automaton class, so group keeps it structure during inheritance.

Create automaton class BasicUser (described on Fig. 3):

```
import java.util.*;
import automaton.*;
public class BasicUser extends DSL{
  public State deleted =
    new State ("Deleted");
  public State suspended =
    new State ("Suspended");
  public StateGroup activation =
    new StateGroup("ActivationG");
  Vector <Entity> activationStates =
    new Vector <Entity> ();
  public Transition
          activation_suspend =
    new Transition(
     "Activation_suspend", "suspend",
     activation, suspended);
  public Transition
          activation_delete =
```

```
    new Transition(
     "Activation_delete", "delete",
     activation, deleted);
  public Transition suspend_passive;
  public Transition suspend_pending;
  public Transition suspend_active;
  void createActive() {
   State stateA = new State(
      "Pending enter mac");
   activationStates.add(stateA);
   State stateB =
       new State ("passive");
   activationStates.add(stateB);
   State stateC = new State (
       "Active enter DA");
   activationStates.add(stateC);
   Transition tran1 = new Transition(
     "passive_pending", "register",
      new Guard() {
        public boolean is() {
          return true;}
      public String toDOT() {
          return "guard";}},
      null, stateB, stateA);
   activationStates.add(tran1);
   Transition tran2 = new Transition(
     "pending_active", "activate",
     new Guard() {
       public boolean is() {
         return true;}
       public String toDOT() {
         return "";}},
       null, stateA, stateC);
   activationStates.add(tran2);
   suspend_passive = new Transition(
     "suspended_passive",
     "unsuspended",
     new Guard() {
     public boolean is() {
       return true;}
     public String toDOT() {
    return "not (guard1 or guard2)";}
  },
  null, suspended, stateA);
  suspend_pending = new Transition(
    "suspended_pending",
    "unsuspended",
    new Guard() {
      public boolean is() {
        return true;}
      public String toDOT() {
        return "guard1"; }
    },
    null, suspended, stateB);
 suspend_active = new Transition(
   "suspended_active",
   "unsuspended",
    new Guard() {
    public boolean is() {
      return true;}
    public String toDOT() {
      return "guard2";}
  },
  null, suspended, stateC);
  activation.setAll(activationStates);
 }
   public BasicUser () {
     super () ;
     createActive () ;
   }
 }
```

Implementation of algorithm for transform to isomorphic graphic model requires additional methods for all classes, used in automaton class. As described in section 2, each component class should contain method for transforming it to DOT. More over this each actions and guard conditions should to be converted to DOT too.

Implementation of the algorithm:

```
public String toString () {
  String res = "";
  String subgraph = "";
// Setup header
  res = "digraph " + "veryuniqname" +
" {\ncompound=true;\n";
// state, groups - extracted using
// reflection from automaton class
// method toString of State,
// Transition and StateGroup convert
// them to DOT representation
  for (State state:states) {
    subgraph+=state.toString(mode); }
  for (StateGroup state:groups) {
    subgraph+=state.toString(mode); }
// get all transitions described in
// class
  for (StateGroup sg:groups) {
transitions.addAll(sg.subtransition);
    getAllSubTransitions(sg);
   }
  for (Transition t:transitions) {
    subgraph += t.toString()+"\n";
  }
// adding border for model
  subgraph  = "subgraph cluster0{\n"
+ subgraph + "\nlabel=\" "+dsl_name+"
\"}";
   res+=subgraph;
   return res;
  }
```

Get DOT text description of automaton class by developed method DSL.saveToFile, which implements algorithm of automatic transfer of code. Figure 4 shows image generated by dot utility from gotten text description.

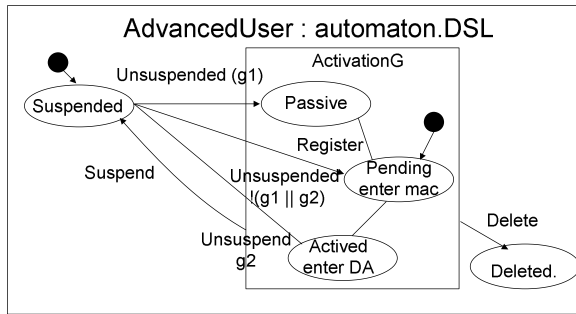

Figure. 4. Graphical model of BasicUser automaton.

5. Analysis of image data obtained by the algorithm of automatically isomorphic transfer

Practical interest is in the reverse engineering of complex inherited automaton classes.

Consider an automatic isomorphic transfer of automaton class AdvancedUser inherited from BasicUser automaton class. Automaton class AdvancedUser is bringing to registration follow functionality:

- to avoid spam registrations user should to type value of CAPTCHA;
- administrator can block user;

To do this, add the following groups:

- Activation – includes group Active, extended by «Captcha» state;
- Deleted – includes two states: «Blocked» – for blocked users and «Deleted» – for deleted users.

Graphical notation for described automaton shown on figure 5:

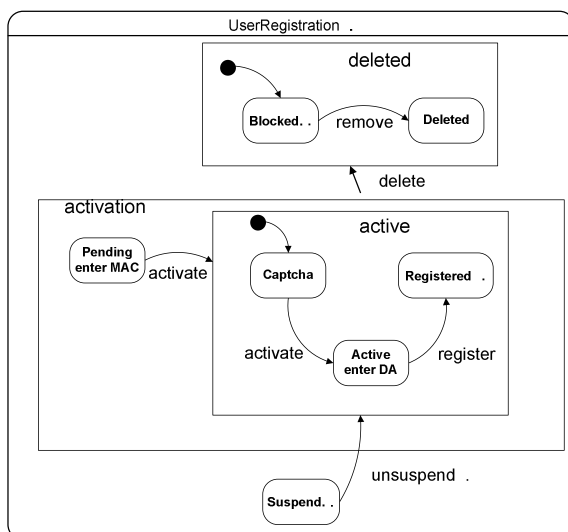

Figure 5. Extended automaton for registration.

Application of the developed algorithm and a method for automatically isomorphic converting for implemented AdvancedUser automaton class provides graphic image presented in figure 6.

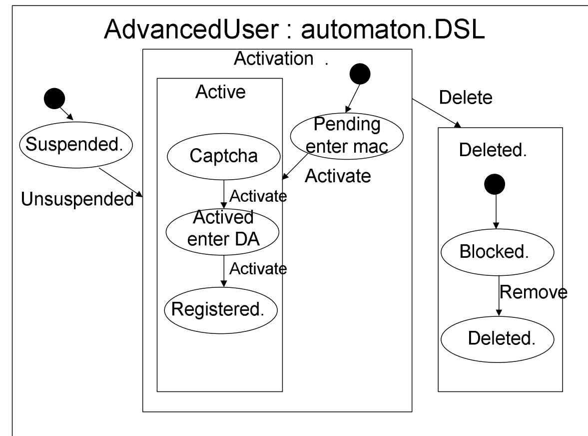

Figure 6. Graphical model of AdvancedUser.

Model obtained by automatic isomorphic transfer (Fig. 6) in line with earlier developed model (Fig. 5)

Thus, the solution of transfer code to graphical isomorphic model problem provides an additional analytical tool for debugging of existing logic, introducing additional functionality or inheritance of automaton classes.

6. Conclusion

Library for automaton classes creation had been developed in this work. The library allows:

- inherit automaton classes and nest groups;
- create methods for states and transitions (also guard conditions);
- eliminate duplication of code and multiply separated methods by using anonymous classes.

Also algorithm of automatic transfer of automaton classes executable code to isomorphic graphical model had been developed. Automaton of user registration from Restful-authentication plug-in developed and described transfer of them into graphical model (Fig. 6). It was shown that the initial model of automaton corresponding to a graphic representation, obtained by library's function for developed automaton class.

The results of this work will be used in further studies:

1) dynamic and static verification of automata displaying counterexamples in visual form, using the reverse engineering;

2) the establishment of libraries, which will facilitate the testing process of automaton classes.

7. References

[1] F. A. Novikov *Visual design of programs*, «Information and control systems». 2005. # 6, http://is.ifmo.ru/works/visualcons/

[2] T Parr. *The Definitive ANTLR Reference: Building Domain-Specific Languages*. Texas: Pragmatic Bookshelf, 2007.

[3] N. I. Polikarpova, A. A. Shalyto *Automaton programming*. SPb.: Piter, 2009. http://is.ifmo.ru/books/_book.pdf

[4] A. A. Astafurov, K. I. Timofeev, A. A. Shalyto, *Automata Classes Inheritance in Dynamic Language Ruby* , Software Engineering Conference (Russia) 2008**.** http://www.secr.ru/etc/secr2008_artyom_astafurov_automata_classes_inheritance_in_ruby.pdf

[5] E. Gansner, E. Koutsofios, S. North.*Drawing graphs with dot*. http://www.graphviz.org

[6] Graphviz package. http://www.graphviz.org

[7] *Academy of Modern Programming.* http://www.amse.ru/courses/oopjava/12.php

[8] E. A Zayakin., A. A. Shalyto, *The method of eliminating repetitive code snippets in the implementation of finite automata*. SPbSU ITMO, 2003. http://is.ifmo.ru/projects/life_app/

[9] D. G. Shopyrin, A. A. Shalyto *The graphical notation of inheritance automaton classes.* «Programming»" 2007. # 5 http://is.ifmo.ru/works/_12_12_2007_shopyrin.pdf