Санкт-Петербургский государственный университет
информационных
технологий, механики и оптики
Кафедра «Компьютерные технологии»

Р.В. Сатюков, И.А. Синев, А.А. Шалыто

Реализация алгоритма *Лампорта* на основе автоматного подхода

Проект выполнен в рамках «Движения за открытую проектную документацию»
http://is.ifmo.ru

Санкт-Петербург
2006

Оглавление

Введение

Цель данной работы – изучение автоматного подхода к программированию с использованием инструментального средства *UniMod*. В качестве примера рассматривается программная реализация автомата для решения задачи взаимного исключения при программировании распределенных систем. Решение рассматриваемой задачи весьма важно, так как оно может быть использовано при распределении общих ресурсов в операционных системах.

1. Формулировка задачи о взаимном исключении

При написании программ, которые состоят из нескольких модулей (например, системы вида клиент-сервер), а также при программировании для распределенных систем [1] часто возникает ситуация, в которой нескольким процессам требуется одновременно получить доступ к некоторым ресурсам.

Получается так, что процессы хотят либо одновременно записать данные в какое-либо место памяти, либо один из них пытается прочитать из памяти, а другой в это время пытается записать какие-то данные в то же самое место памяти.

Для разрешения подобных конфликтов между процессами используются алгоритмы взаимного исключения (*mutual exclusion algorithms* или *mutex algorithms*). В частности, эти алгоритмы применяются в параллельном программировании для того, чтобы избежать одновременного использования несколькими процессами *un-shareable* ресурсов – таких ресурсов, которые могут быть использованы одновременно только одним процессом.

Рассмотрим простой пример.

Пусть клиент хочет снять со счета в банке 100 рублей. В этом случае банк действует следующим образом.

1. *Убеждается, что на счете клиента есть как минимум 100 рублей.*
2. *Снимает 100 рублей со счета клиента.*

Но что произойдет, если между этими действиями кто-то (второй клиент) успел снять с этого счета деньги?

Предположим, что на счету было 150 рублей. Рассмотрим следующую последовательность действий.

1. *Банк выполняет проверку наличия средств для первого клиента.*
2. *Банк выполняет ту же операцию для второго клиента.*
3. *Банк снимает со счета 100 рублей по запросу второго клиента.*
4. *Банк снимает со счета 100 рублей по запросу первого клиента.*

В итоге, на счете в банке осталось минус 50 рублей.

Простейшим методом взаимного исключения на однопроцессорном компьютере может быть запрещение прерываний при выполнении некоторых важных частей программы (*critical sections*).

Критическая секция – это фрагмент программы, который может исполняться не более чем одним процессом (или потоком) одновременно.

Таким образом, при выполнении фрагмента программы, находящегося в критической секции, никаких конфликтов между процессами не будет – в этой секции может находиться только один процесс.

Системы, состоящие из нескольких процессов, часто легче программировать с применением критических секций. Когда процессу требуется читать или модифицировать какие-либо разделяемые структуры данных, он входит в критическую секцию. Этим он обеспечивает себе исключительное право использования этих данных, и можно быть уверенным, что никакой другой процесс не будет иметь доступа к этому ресурсу одновременно с ним.

2. Разделяемые переменные

Параллельные программы сильно зависят от разделяемых компонентов, поскольку процессы могут работать над одной задачей только взаимодействуя между собой. Единственный способ взаимодействия — возможность для одного процесса записывать данные куда-то, откуда другой процесс их читает. Это может быть разделяемая переменная. Поэтому взаимодействие может программироваться, как запись и чтение разделяемых переменных.

Программы с разделяемыми переменными чаще всего выполняются на машинах с разделяемой памятью, поскольку в них каждый процесс может получить непосредственный доступ к каждой переменной. Однако программирование с разделяемыми переменными можно использовать и в машинах с распределенной памятью [1].

3. Алгоритм *Петерсона* для двух потоков

Одним из наиболее простых алгоритмов, предназначенных для выполнения операций с критическими секциями, является алгоритм *Петерсона* [2]. На псевдокоде он имеет вид:

```
1: void lock(int i) {
2:     int j = 1 - i;
3:     want[i] = true;
4:     turn = j;
5:     while (want[j] && (turn == j)) ;
6: }

1: void unlock(int i) {
```

```
2:      want[i] = false;
3: }
```

Отметим, что в алгоритме используется две процедуры: процедура `lock`, которая отвечает за вход в критическую секцию, а процедура `unlock` – за выход из нее.

Переменные `want[0]` и `want[1]` содержат информацию о том, «хочет» ли соответствующий процесс войти в критическую секцию.

Алгоритм *Петерсона* обладает следующими свойствами:

1. *Взаимное исключение:* два процесса не могут находиться в критической секции одновременно.
2. *Прогресс:* если один из процессов пытается войти в критическую секцию, и она не занята другим процессом, то какой-то из процессов ее получит.
3. *Отсутствие голодания:* если процесс хочет попасть в критическую секцию, он в нее когда-нибудь попадет.

Известно обобщение этого алгоритма на случай большего числа потоков. Однако в этом случае алгоритм имеет более сложный вид. Кроме того, в этом алгоритме не выполняется свойство *честности* [1] – процессорное время может распределяться между потоками неравномерно.

4. Параллельное программирование и конечные автоматы

Выполнение параллельной программы можно представить как чередование атомарных (*atomic*) операций.

Атомарная операция — это операция, выполнение которой является неделимым действием. В любой «момент времени» атомарная операция или была уже выполнена или ее выполнение еще предстоит (она выполняется как бы «мгновенно»).

Конечный автомат является достаточно простой и удобной моделью вычислений. С помощью конечных автоматов можно реализовать алгоритмы параллельного программирования, при этом **действия, выполняемые автоматом при переходе из одного состояния в другое, будут являться атомарными**. Соответственно, чем более сложные операции будут сделаны атомарными, тем проще станет получившийся автомат. С другой стороны, чем более простыми будут атомарные операции, тем большую гибкость будет иметь алгоритм.

Таким образом, при использовании конечных автоматов могут быть явно выделены все операции, которые являются атомарными.

5. Алгоритм *Лампорта*

Алгоритм *Лампорта* (*алгоритм булочной, Lamport's bakery algorithm*) решает задачу взаимного исключения для ***N*** процессов. Он был впервые описан в статье [3]. Основная идея алгоритма позаимствована из принципа работы магазина: процессы (по аналогии с покупателями) выбирают себе номера, а затем процесс, обладающий наименьшим номером, входит в критическую секцию.

Рассмотрим этот метод более подробно. Представим себе булочную, где при входе стоит автомат, который каждому приходящему клиенту выдает листочек с написанным на нем номером. При этом номер увеличивается с приходом каждого нового клиента. Каждый раз, когда продавец оказывается свободным, он обслуживает клиента с наименьшим номером.

Приведем пример возможной последовательности действий.

Новый клиент (A) заходит в булочную. Автомат выдает ему номер 1.
Новый клиент (B) заходит в булочную. Автомат выдает ему номер 2.
Продавец освобождается и обслуживает клиента A (в этот момент в булочной находятся клиенты с номерами 1 и 2, наименьший из этих номеров – один).
Новый клиент (C) заходит в булочную. Автомат выдает ему номер 3.
Продавец освобождается и обслуживает клиента B (в этот момент в булочной находятся клиенты с номерами 2 и 3, наименьший из этих номеров – два).
Продавец освобождается и обслуживает клиента C (в этот момент в булочной находится только клиент с номером 3).
Новый клиент (D) заходит в булочную. Автомат выдает ему номер 4.
Продавец освобождается и обслуживает клиента D (в этот момент в булочной находится только клиент с номером 4).
Новый клиент (E) заходит в булочную. Автомат выдает ему номер 5.
Новый клиент (F) заходит в булочную. Автомат выдает ему номер 6.
Новый клиент (G) заходит в булочную. Автомат выдает ему номер 7.
Продавец освобождается и обслуживает клиента E.
Продавец освобождается и обслуживает клиента F.
Новый клиент (H) заходит в булочную. Автомат выдает ему номер 8.
Продавец освобождается и обслуживает клиента G.
Продавец освобождается и обслуживает клиента H.

В этой модели все действия происходили последовательно. В реальности же может случиться так, что два клиента придут одновременно. Какие номера им выдать в этом случае?

Одно из возможных решений этой проблемы состоит в следующем: выдадим им одинаковые номера, и дадим продавцу указание при выборе из двух клиентов с одинаковым номером обслуживать того клиента, у которого

меньше номер паспорта. Таким образом, считается, что все клиенты как-то упорядочены по приоритету.

Алгоритм *Лампорта* действует аналогичным способом. При этом роль клиентов играют потоки, а обслуживание продавцом соответствует действиям процесса в критической секции. Аналогом номера паспорта в случае взаимодействия потоков будет служить идентификатор потока.

Приведем реализацию данного алгоритма на псевдокоде, в том виде, как она приведена в книге по параллельному и распределенному программированию [4]:

```
 1: void lock(int i) {
 2:     choosing[i] = true;
 3:     for (int j = 0; j < n; j++) {
 4:         if (number[j] >= number[i])
 5:             number[i] = number[j];
 6:     }

 7:     number[i]++;
 8:     choosing[i] = false;

 9:     for (int j = 0; j < n; j++) {
10:         while (choosing[j]);
11:         while ((number[j] <> 0) && ((number[j] < number[i])
12:         || ((number[j] == number[i]) && (j < i))));
13:     }
14: }
```

```
 1: void unlock(int i) {
 2:     number[i] = 0;
 3: }
```

Можно доказать, что алгоритм работает даже если возможны чтение и запись некоторого сегмента памяти несколькими процессами.

Приведенная запись алгоритма трудна для понимания. Поэтому ниже рассматривается применение автоматов для реализации этого алгоритма. При этом рассматриваются два метода построения графов переходов: эвристический и формальный переход от схемы алгоритма. Выполнена также декомпозиция и оптимизация графа переходов, полученного вторым методом

6. Автомат, построенный эвристическим методом

Одним из возможных методов построения автомата по программе является эвристический. Он не может считаться надежным, однако позволяет сохранить логическую структуру программы. На рис. 1, 2 приведен один из возможных вариантов схемы связей и графа переходов автомата.

Для построения программы с использованием конечных автоматов применяется инструментальное средство *UniMod* http://unimod.sourceforge.net/intro.html. Для хранения данных и отображения информации, а также для обеспечения связи с системой пользовательского интерфейса к автомату подключаются объекты управления. В данном случае это экземпляры классов *Process* и *ProcessPane*.

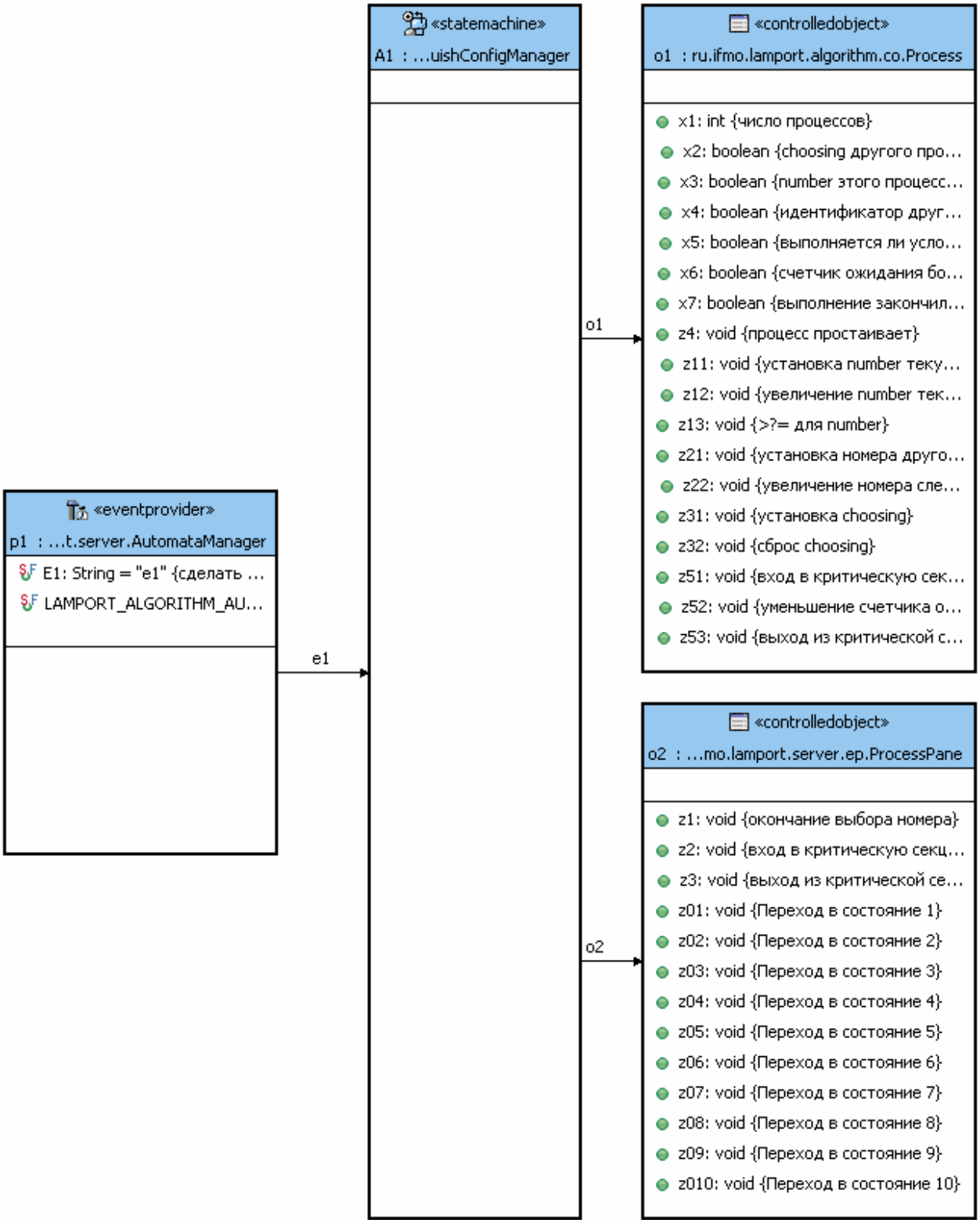

Рис.1. Схема связей автомата *A1*

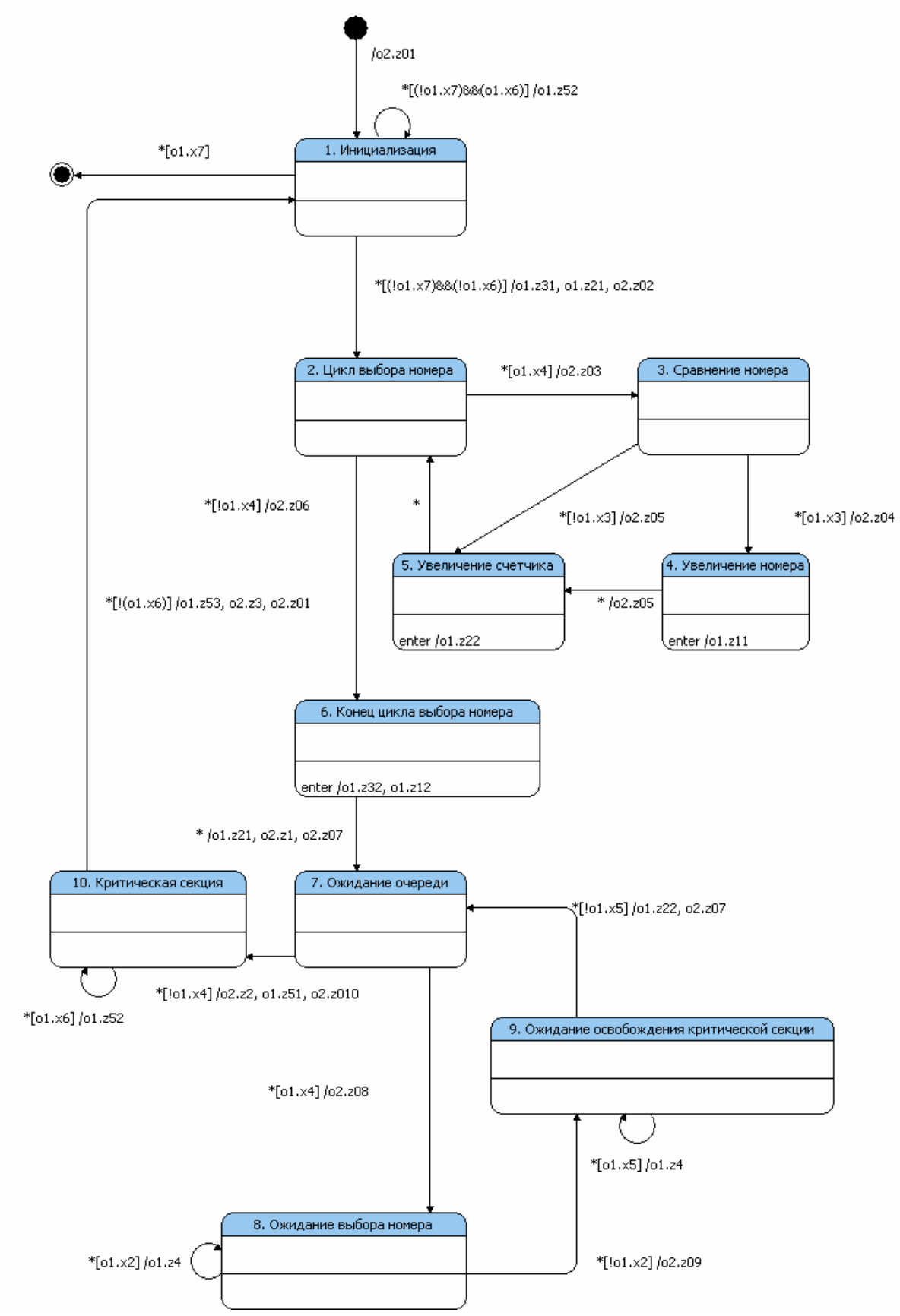

Рис.2. Граф переходов *A1*, построенный эвристическим методом

7. Формальный метод построения графа переходов

7.1. Построение схемы алгоритма

Первым шагом построения графа переходов по данному алгоритму является построение схемы алгоритма (рис.3) по псевдокоду, который приведен в разд. 5.

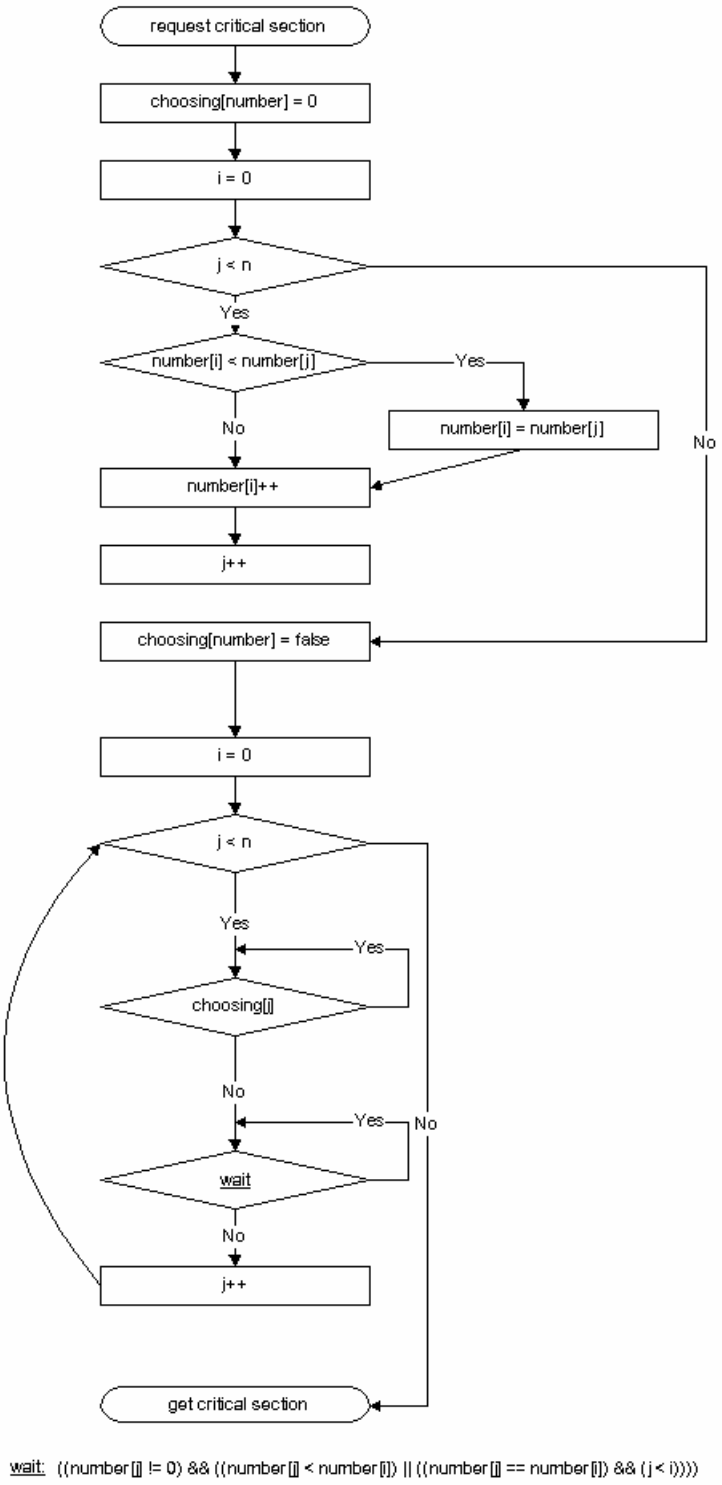

Рис.3. Схема алгоритма

7.2. Реализация алгоритма Лампорта с использованием одного графа переходов

В соответствии с методом, описанным в работе [5], схема алгоритма, изображенная на рис. 3, преобразуется в граф переходов (рис. 4). При этом схема связей автомата не изменяется (рис. 1).

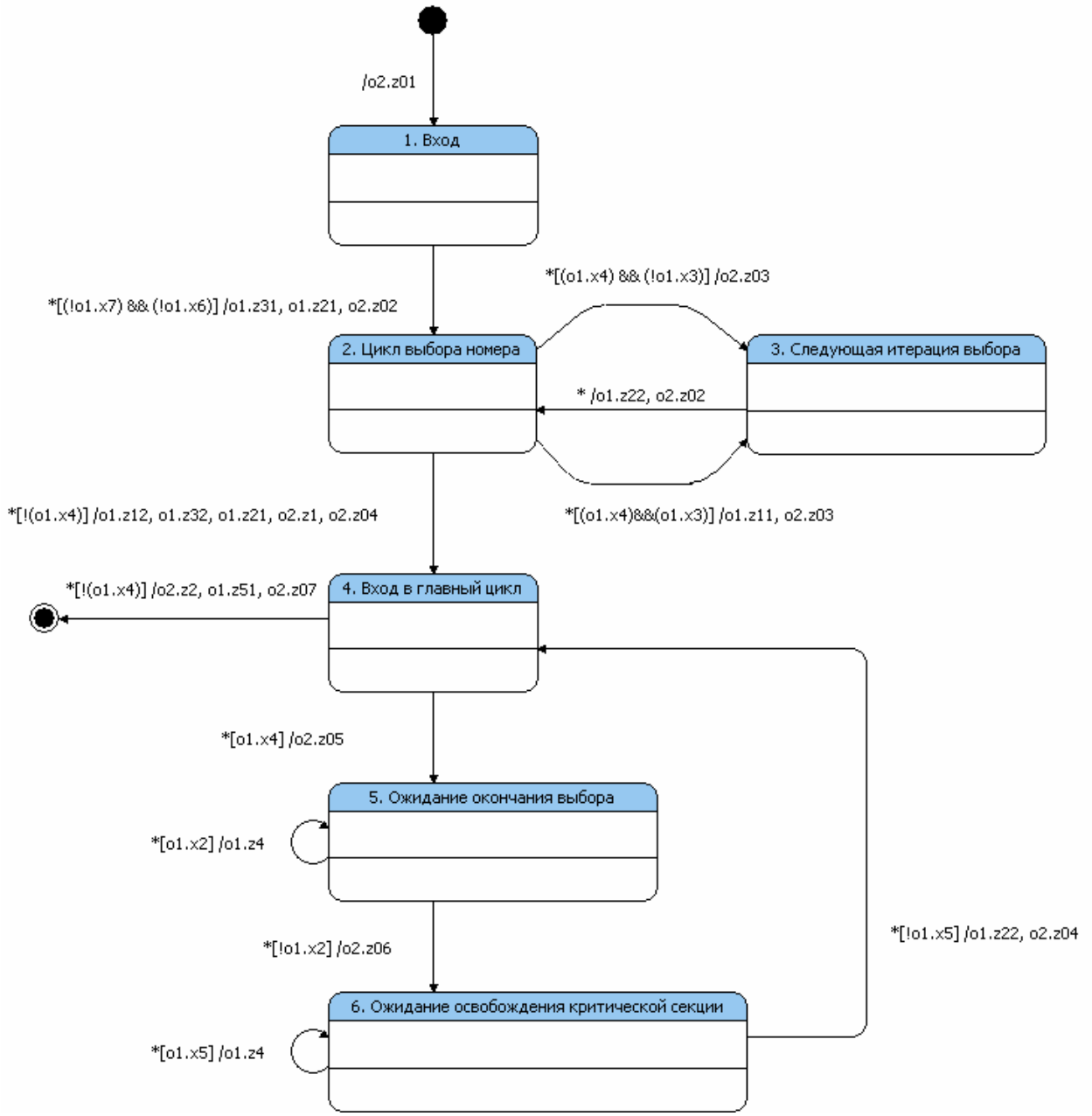

Рис. 4. Граф переходов полученный по схеме алгоритма

Однако целью настоящей работы является построение автомата, который эмулировал бы обращение процесса к критической секции в течение длительного времени с некоторым периодом. Это выполняется добавлением состояния «Критическая секция», соответствующего определенным действиям после входа в нее, и перехода из состояния «Вход» в себя, отвечающего за независимые от критической секции действия процесса. В обоих случаях ожидание прекращается, когда соответствующий счетчик

уменьшается до нуля. Адаптированный к решаемой задаче описанным образом граф переходов изображен на рис. 5.

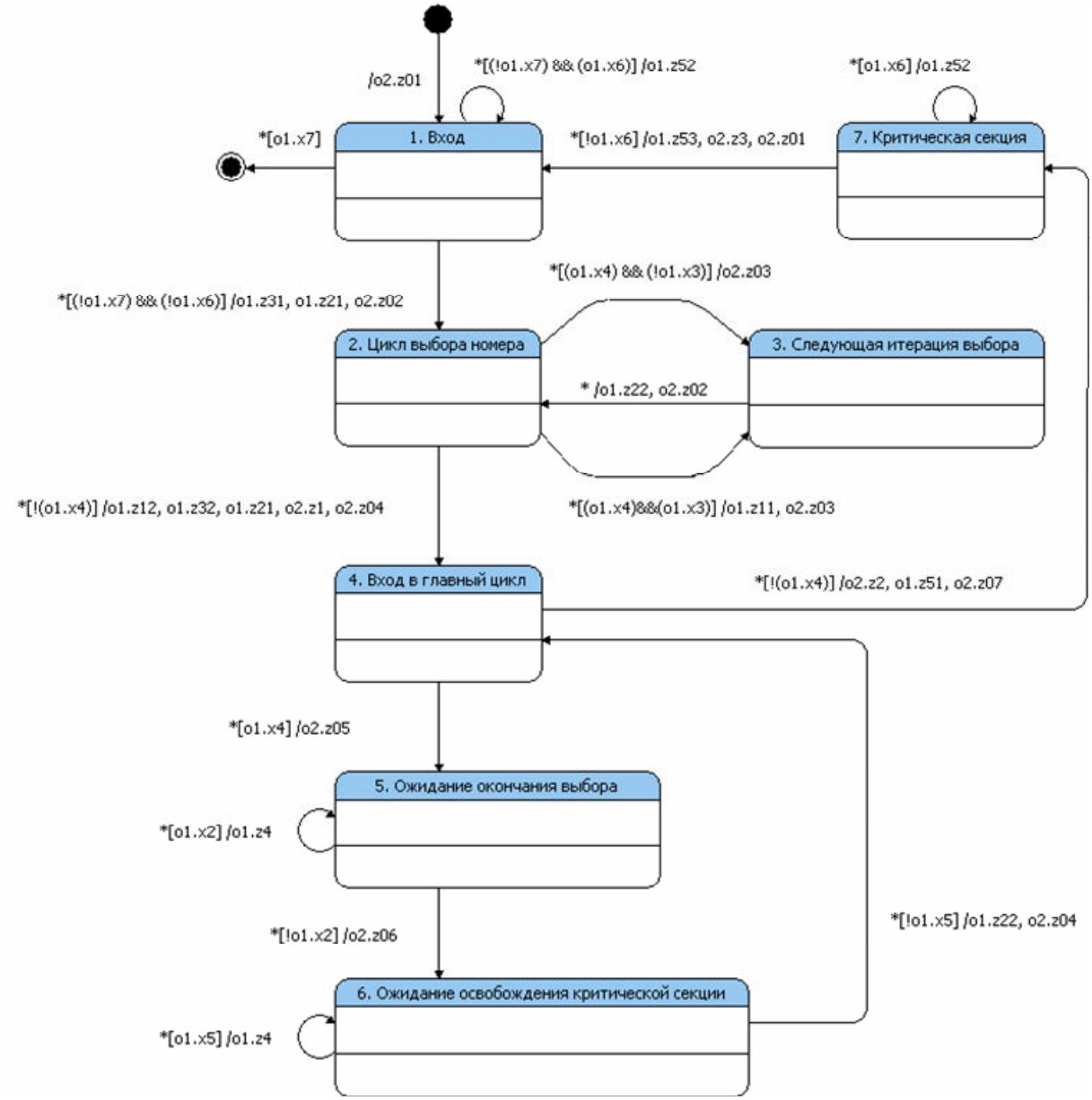

Рис. 5. Граф переходов *A2*, полученный по модифицированной схеме алгоритма

Полученный автомат модифицирован с учетом решаемой задачи визуализации путем добавления состояния *Critical section*. Это состояние моделирует поведение процесса, когда он находится в критической секции. В данном случае автомат не совершает никаких действий.

7.3. Оптимизация путем введения дополнительной операции

Введение дополнительной атомарной операции «Взятие максимума из двух чисел» позволяет уменьшить автомат на одно состояние. Полученный таким образом автомат изображен на рис. 6. Такое преобразование не затрагивает схему связей.

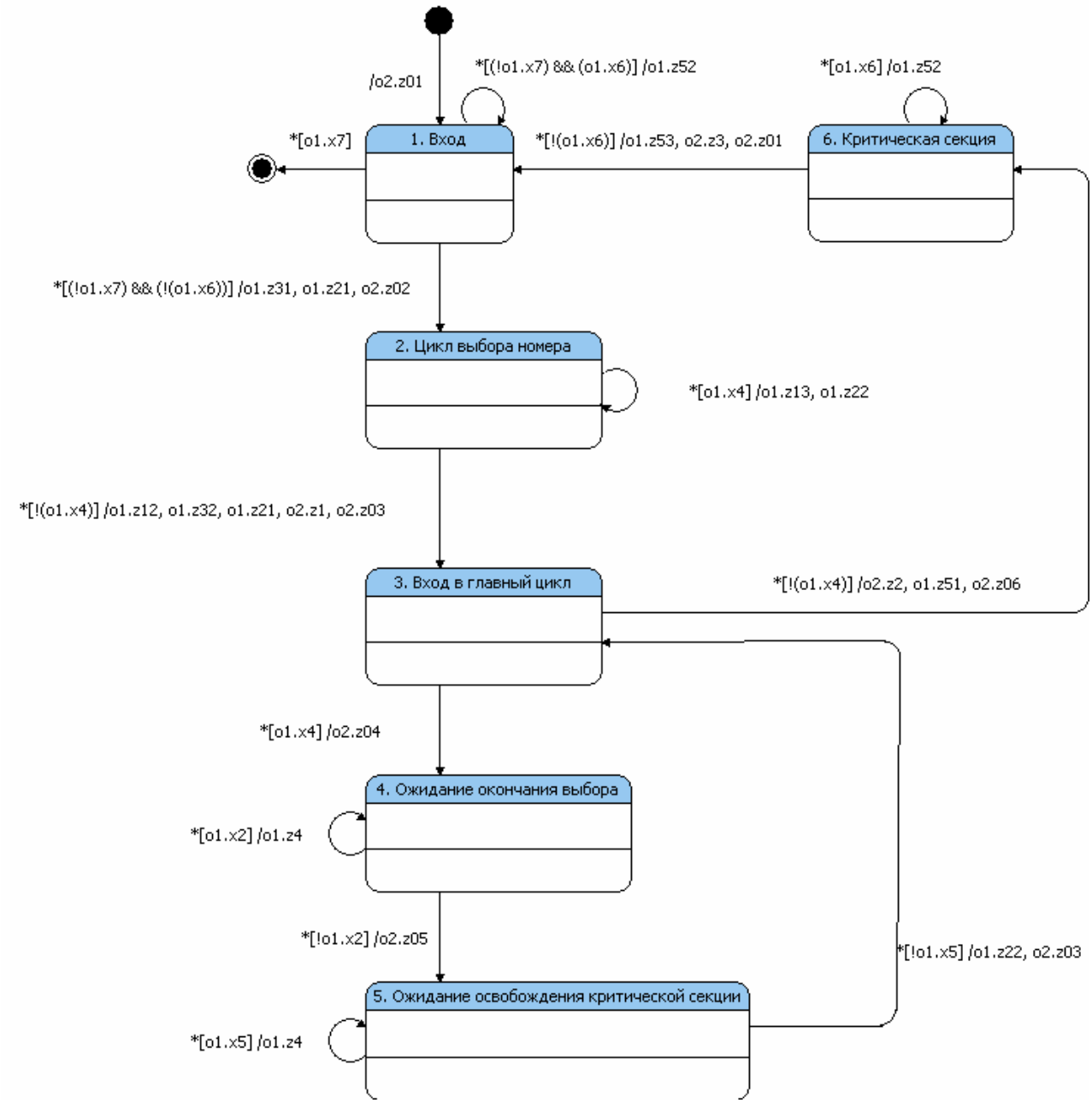

Рис.6. Граф переходов *A3*, упрощенный за счет введения атомарной операции взятия максимума

Корректность данного преобразования подтверждается правильным взаимодействием исходного и модифицированного автоматов.

8. Структура программы

8.1. Управляющий автомат

В предыдущем разделе были рассмотрены различные способы преобразования алгоритма *Лампорта* в конечный автомат. Следует обратить внимание, что **каждому из процессов соответствует свой автомат**, контролирующий выделение времени в критической секции данному

процессу. Теперь необходимо описать способ, которым автоматы взаимодействуют между собой.

Итак, программа должна состоять из двух частей (рабочей и серверной).

Рабочая часть содержит в себе конечный автомат, реализующий алгоритм *Лампорта*, а также обеспечивает взаимодействие этого автомата с серверной частью.

Серверная часть моделирует процесс доступа нескольких потоков к критической секции, поддерживая взаимодействие отдельных автоматов. В ней же выполняется управление параметрами моделирования.

Взаимодействие между основным (серверным) автоматом и автоматами процессов, выполняется введением особого объекта управления (*AutomataManager*). Он контролируется серверным автоматом и отвечает за создание и отправку сообщений управляемым автоматам.

Автоматы, соответствующие процессам, управляются главным автоматом – *Server*, схема связей и диаграмма переходов которого изображены на рис.7,8. Этот автомат обрабатывает события, генерируемые таймером и пользовательским интерфейсом, а так же инициирует выполнение операций автоматами процессов.

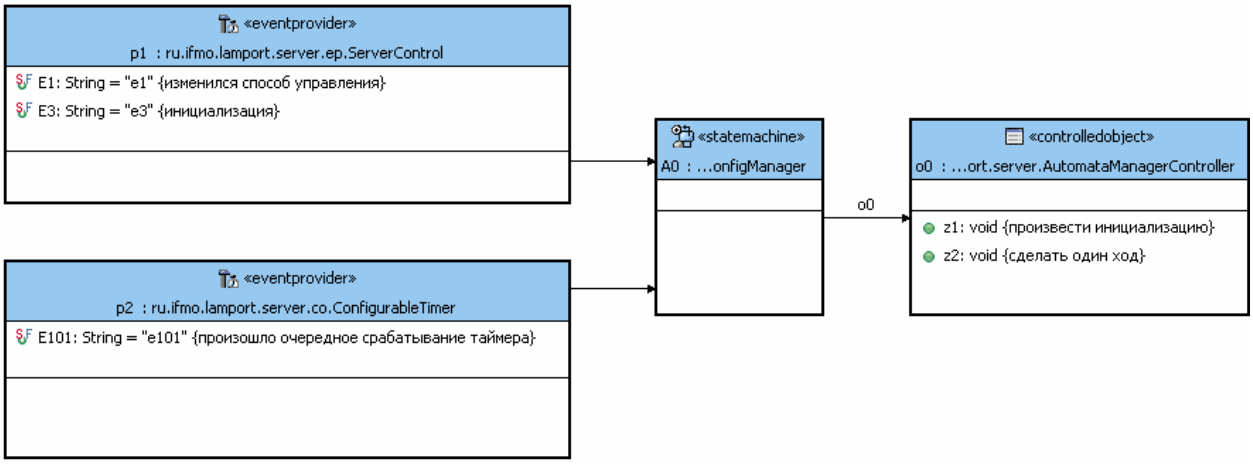

Рис.7. Схема связей главного автомата

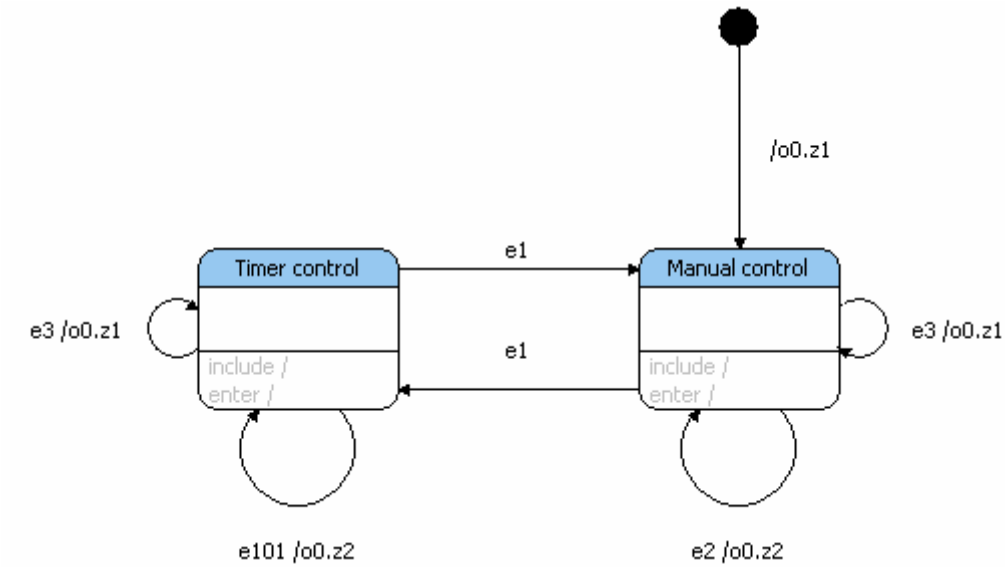

Рис.8. Диаграмма состояний главного автомата

8.2. Общая структура программы

Теперь, когда рассмотрены две основные части программы, можно описать взаимодействие между ними. Объекту *AutomataNamagerController,* который управляется главным автоматом, подчиняется объект *AutomataManager*, отвечающий за создание и непосредственное управление автоматами процессов. Диаграмма связей системы изображена на рис. 9.

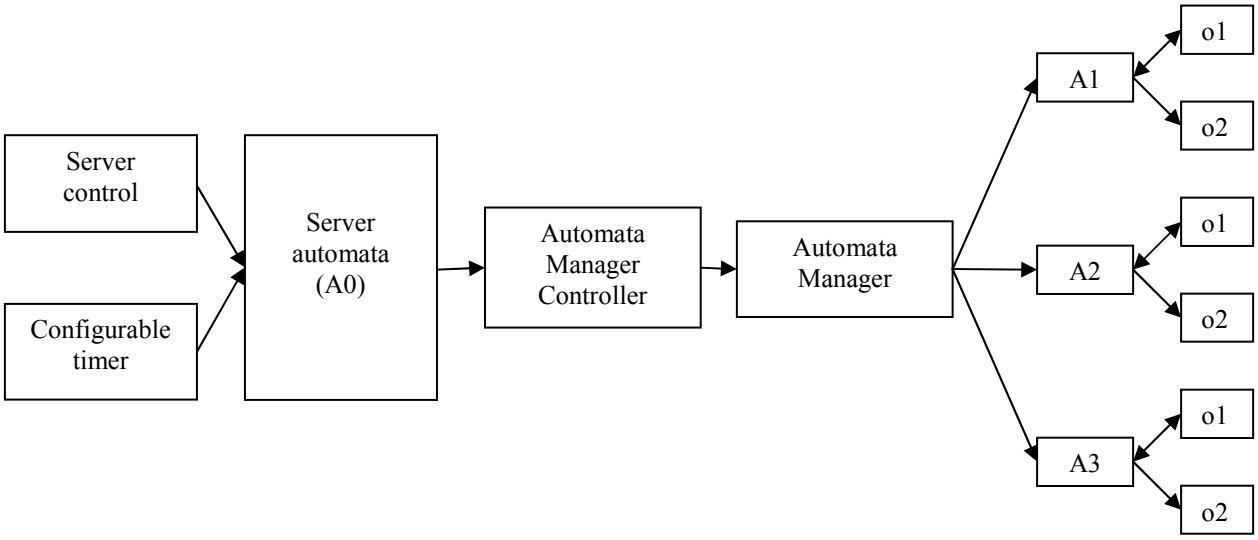

Рис. 9. Структура программы

9. Пользовательский интерфейс

Программа отображает взаимодействие процессов в двух режимах: глобальном, в котором отображаются только ключевые изменения состояния системы, и подробном, в котором фиксируется каждый этап.

При этом реализовано много возможностей для гибкого управления распределением времени. Кроме того, обеспечено добавление и удаление процессов, а также переключение между ручным управлением и управлением по таймеру. На рис. 10 – 12 изображен вид окна в разных режимах.

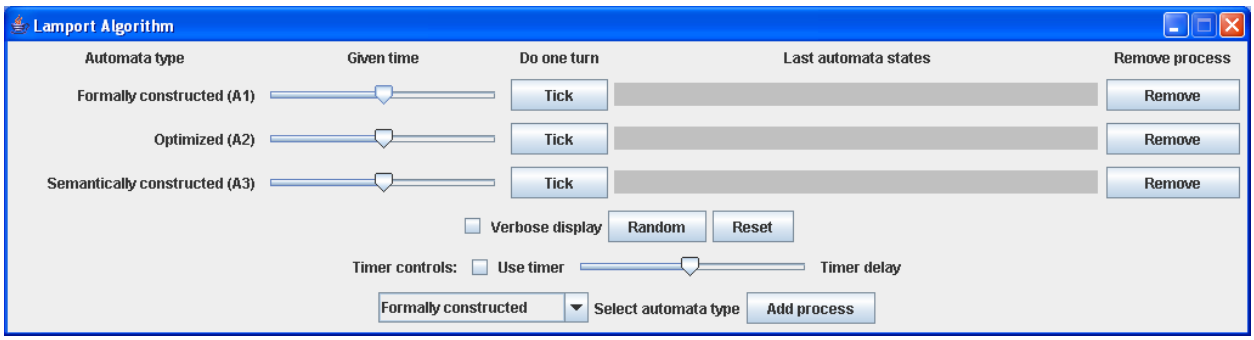

Рис.10. Вид окна сразу после запуска

Окно состоит из двух основных частей: панели управления параметрами системы в целом и нескольких панелей, отвечающих за управление конкретными процессами.

Панель глобального управления содержит следующие элементы:

- флажок использования таймера (*Use timer*) позволяет включить или выключить генерацию событий по таймеру;
- флажок переключения режима демонстрации локальных и глобальных состояний (*Verbose display*), при установленном флажке подробно показываются изменения состояния процессов;
- кнопка выделения такта времени случайному процессу (*Random*) дает такт времени случайно выбранному процессу, при этом соответствующий автомат делает один переход;
- ползунок изменения задержки таймера (*Timer delay*) позволяет регулировать период, с которым генерируются события таймера, при движении ползунка вправо происходит увеличение периода и уменьшение скорости;
- выпадающий список выбора типа добавляемого автомата (*Select automata type)* позволяет выбрать тип автомата, используемого для контроля добавляемого процесса;
- кнопка добавления автомата (*Add process*) добавляет новый процесс с управляющим автоматом выбранного типа.

Панель-строка, соответствующая каждому из процессов состоит из следующих элементов:

- тип автомата, связанного с данным процессом;
- ползунок управления выделяемым процессу временем (*Given time*) – позволяет регулировать относительное количество времени, выделяемого данному процессу;
- кнопка выделения одного такта времени (*Tick*) дает один такт времени данному процессу;
- диаграмма состояний автомата (*Last automata states*) показывает последние состояния автомата. Каждое состояние отображается прямоугольником. Одновременные состояния разных автоматов показываются друг под другом. Состояния зеленого цвета относятся к выбору номера, желтого – к процессу ожидания, а красного – к нахождению в критической секции. Внутри прямоугольника написан номер состояния, а так же номер, выбранный соответствующим процессом, если он есть;
- кнопка удаления процесса (*Remove*) удаляет соответствующий автомату процесс из системы при этом вся система перезапускается, как того требует алгоритм.

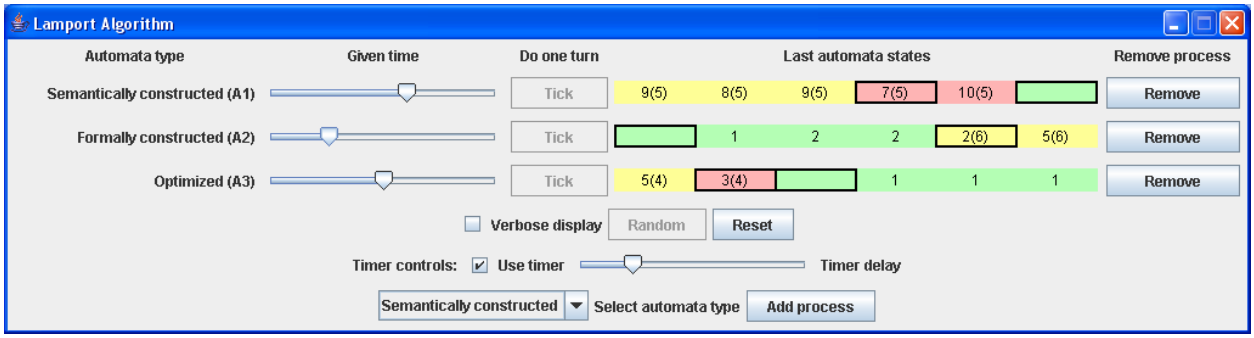

Рис. 11. Режим демонстрации глобального состояния

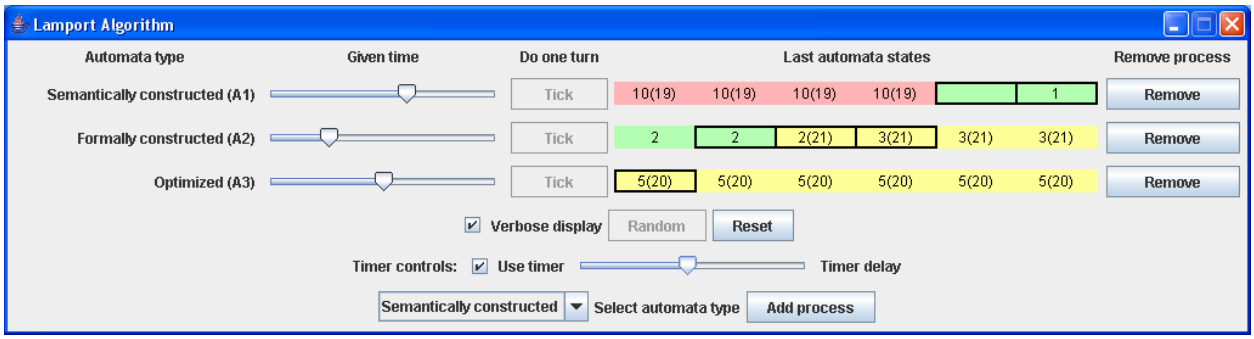

Рис. 12. Режим демонстрации локальных состояний

Заключение

Данная программа совместно с приведенными графами переходов позволяет по-новому взглянуть на алгоритм *Лампорта*. При этом наглядно отображаются:

- стадии, через которые проходит с течением времени система борющихся за критическую секцию потоков;
- изменения, происходящие под влиянием способа выделения потокам процессорного времени.

Инструментальное средство *Unimod* позволяет визуально строить автоматы и схемы связей. Это значительно упрощает применение такого метода на практике, и делает более прозрачным принцип построения программы.

Для внедрения автоматов в рассматриваемой области, необходимо, чтобы разработчики операционных систем и библиотек параллельного программирования использовали их уже при построении моделей исполняющей среды. Эта среда должна обеспечивать достаточную мощность атомарных операций вместе с возможностью их практической реализации.

Литература

1. *Эндрюс Г. Р.* Основы многопоточного, параллельного и распределенного программирования. М.: Вильямс, 2003.
2. *Peterson G.L.* Myths about the mutual exclusion problem //Inform. Process. Lett. 12 (3) (1981), pp. 115, 116.
3. *Lamport L.* A new solution of Dijkstra's Concurrent Programming Problem // Comm. ACM. 17, 8 (August 1974), 453 – 455.
4. *Garg V. K.* Concurrent and Distributed Computing in Java. Wiley-IEEE Press, 2004.
5. *Шалыто А.А., Туккель Н.И.* Преобразование итеративных алгоритмов в автоматные //Программирование. 2002. № 5, с. 12–26. http://is.ifmo.ru/works/iter/
6. *Börger E.* Specification and Validation Methods. Oxford University Press, 1995.

Приложение. Исходные коды на *Java*

Start.java

Start.java

```
package ru.ifmo.lamport;

import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.IOException;

import org.apache.commons.logging.LogFactory;
import org.apache.log4j.xml.DOMConfigurator;

import ru.ifmo.lamport.server.AutomataManager;

import com.evelopers.common.exception.CommonException;
import com.evelopers.unimod.adapter.standalone.Run;
import com.evelopers.unimod.core.stateworks.Model;
import com.evelopers.unimod.debug.ExceptionHandlerImpl;
import com.evelopers.unimod.runtime.EventProcessorListener;
import com.evelopers.unimod.runtime.ExceptionHandler;
import com.evelopers.unimod.runtime.ModelEngine;
import com.evelopers.unimod.runtime.interpretation.InterpretationHelper;
import com.evelopers.unimod.runtime.logger.SimpleLogger;
import com.evelopers.unimod.transform.xml.XMLToModel;

public class Start {

    private static final String MAIN_AUTOMATA_XML = "A0.xml";


  public static void main(String[] args) {

        try {
          DOMConfigurator.configure(Run.class.getResource("log4j.xml"));
          InterpretationHelper helper = InterpretationHelper.getInstance();
          Model model = XMLToModel.loadAndCompile(
              new FileInputStream(AutomataManager.RESOURCES_DIRECTORY
                  + MAIN_AUTOMATA_XML)
            );
          ModelEngine engine = helper.createStandAloneModelEngine(model, true
);
          EventProcessorListener logger = new SimpleLogger(
              LogFactory.getLog(Run.class));
            engine.getEventProcessor().addEventProcessorListener(logger);
            ExceptionHandler eh = new ExceptionHandlerImpl();
            engine.getEventProcessor().addExceptionHandler(eh);
            engine.start();
        } catch(CommonException e) {
          e.printStackTrace();
        } catch (FileNotFoundException e) {
      e.printStackTrace();
    } catch (IOException e) {
      e.printStackTrace();
    }

    }

}
```

AutomataManager.java

```
package ru.ifmo.lamport.server;

import java.io.FileInputStream;
import java.io.IOException;
import java.util.ArrayList;

import org.apache.commons.logging.LogFactory;

import ru.ifmo.lamport.algorithm.co.Process;
import ru.ifmo.lamport.algorithm.co.ServerData;
import ru.ifmo.lamport.server.ep.MainDialog;
import ru.ifmo.lamport.server.ep.ProcessPane;

import com.evelopers.common.exception.CommonException;
import com.evelopers.unimod.adapter.standalone.Run;
import com.evelopers.unimod.core.stateworks.Event;
import com.evelopers.unimod.core.stateworks.Model;
import com.evelopers.unimod.debug.ExceptionHandlerImpl;
import com.evelopers.unimod.runtime.ControlledObject;
import com.evelopers.unimod.runtime.ControlledObjectsMap;
import com.evelopers.unimod.runtime.EventProcessorListener;
import com.evelopers.unimod.runtime.EventProvider;
import com.evelopers.unimod.runtime.ExceptionHandler;
import com.evelopers.unimod.runtime.ModelEngine;
import com.evelopers.unimod.runtime.context.StateMachineContextImpl;
import com.evelopers.unimod.runtime.interpretation.InterpretationHelper;
import com.evelopers.unimod.runtime.logger.SimpleLogger;
import com.evelopers.unimod.transform.TransformException;
import com.evelopers.unimod.transform.xml.XMLToModel;

/**
 * @author Igor Sinev
 */
public class AutomataManager implements EventProvider {

  public static final String RESOURCES_DIRECTORY = "resources/";

  public static final int DEFAULT_PROCESS_COUNT = 3;
  private static final String[] AUTOMATA_FILES = new String[] {
                            "A1.xml",
                            "A2.xml",
                            "A3.xml"
                          };

    public static ModelEngine[] engines;
    private static int processCount = DEFAULT_PROCESS_COUNT;

    public static int getProcessCount() {
      return processCount;
    }

    public static void setProcessCount(int n) {
      processCount = n;
    }

    public static ArrayList<Integer> automataModelList
        = new ArrayList<Integer>();

    public AutomataManager() {
      for (int i = 0; i < 3; i++)
        automataModelList.add(i);
    }
```

```
static Process[] processes;

public static int getAutomataNumber(int i) {
  return ServerData.getNumber(i);
}

public void initProcessAutomata() {

  InterpretationHelper helper = InterpretationHelper.getInstance();

    engines = new ModelEngine[processCount];

    assert(processCount == automataModelList.size());

    processes = new Process[processCount];

    for (int i = 0; i < processCount; i++) {
        Model model;
        try {
            model = XMLToModel.loadAndCompile(
                new FileInputStream(
                    RESOURCES_DIRECTORY
                    + AUTOMATA_FILES[automataModelList.get(i)]
                )
              );
        } catch (IOException e) {
            System.out.println("[ERROR]   Can't load xml: "
                + RESOURCES_DIRECTORY
                + AUTOMATA_FILES[automataModelList.get(i)]
              );
            throw new RuntimeException();
        } catch (TransformException e) {
            System.err.println(e.getMessage());
            throw new RuntimeException();
        }

        ModelEngine processEngine;

        try {
            Process p = new Process(i);
            processes[i] = p;
            ProcessPane pane = new ProcessPane(i);
            MainDialog.addProcessPane(pane);
            processEngine = helper.createBuildInModelEngine(
                model,
                new ProcessCOMap(p, pane),
                true
              );

            p.setModelEngine(processEngine);
        } catch (CommonException e) {
            System.err.println(e.getMessage());
            throw new RuntimeException();
        }

        EventProcessorListener logger = new SimpleLogger(
            LogFactory.getLog(Run.class));
        processEngine.getEventProcessor().addEventProcessorListener(
                                                    logger);

        ExceptionHandler eh = new ExceptionHandlerImpl();
        processEngine.getEventProcessor().addExceptionHandler(eh);
```

```
            engines[i] = processEngine;
        }
    }

    public void init(ModelEngine engine) throws CommonException {

    }

    public void sendEventToProcess(int pNum, String event) {
      synchronized (AutomataManager.class) {
            System.out.println("[EVENT]    Sending event "
                + event
                + " to process number " + pNum);
          if (pNum < 0)
            return;

            MainDialog.instance.nothingHappened = true;
            engines[pNum].getEventManager().handleAndWait(new Event(event),
                    StateMachineContextImpl.create());
            if (MainDialog.instance.nothingHappened
                && MainDialog.instance.lastVerboseDisplayValue) {
              MainDialog.instance.updateAllExcept(-1);
              MainDialog.instance.setLastActive(pNum);
            }
    }
    }

    public void doInit() {
        initProcessAutomata();

        for (int i = 0; i < getProcessCount(); i++) {
            sendEventToProcess(i, "e1");
        }
        System.out.println("Init is done");
    }

    private int stepNumber = 0;

/**
 * @unimod.event.descr сделать следующий шаг
 */
public static final String E1 = "e1";
    public void doNextStep() {
      stepNumber++;

      sendEventToProcess(MainDialog.instance.getRandomNumberGenerator()
          .randomProcessNumber(), Process.NEXT_TICK);
      MainDialog.instance.repaint();
    }

    /**
     * Controlled object map implementation for
     * process. Returns controlled object,
     * which is specified when constructing.
     *
     * @author Igor Sinev
     */
    private static class ProcessCOMap implements ControlledObjectsMap {

        private ControlledObject processCo;
        private ControlledObject displayCo;

        public ProcessCOMap(ControlledObject processCo,
                  ControlledObject displayCo) {
```

```
            this.processCo = processCo;
        this.displayCo = displayCo;
      }

      public ControlledObject getControlledObject(String coName) {
            if (coName.equals("o1"))
              return processCo;
            else if (coName.equals("o2"))
              return displayCo;
            else
              throw new RuntimeException(
                  "Illegal object requested from the object map");
          }
      }

    public void dispose() {
    }

    public void addAutomata(int i) {
      synchronized (AutomataManager.class) {
        processCount++;
        automataModelList.add(i);
      }
    }

    public void removeAutomata(int i) {
      synchronized (AutomataManager.class) {
        processCount--;
        automataModelList.remove(i);
      }
    }
}
```

AutomataManagerController.java

```
package ru.ifmo.lamport.server;

import javax.swing.JFrame;

import ru.ifmo.lamport.server.ep.MainDialog;

import com.evelopers.unimod.runtime.ControlledObject;
import com.evelopers.unimod.runtime.context.StateMachineContext;

/**
 * Controlled object for Server automata that
 * is responcible for interaction with process automata.
 */
public class AutomataManagerController implements ControlledObject {

    private AutomataManager manager;

    public AutomataManagerController() {
      manager = new AutomataManager();
    }

    boolean firstCall = true;
    /**
     * @unimod.event.descr сделать один ход
     */
    public static final String E1 = "e1";

    /**
     * @unimod.action.descr произвести инициализацию
```

```
   */
  public void z1(StateMachineContext context) {
    manager.doInit();

    MainDialog.instance.doInitialization();

    MainDialog.instance.manager = manager;
      MainDialog.instance.pack();
      MainDialog.instance.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      MainDialog.instance.setResizable(false);
      MainDialog.instance.setVisible(true);
  }

  /**
   * @unimod.action.descr сделать один ход
   */
  public void z2(StateMachineContext context) {
    manager.doNextStep();
  }

}
```

ConfigurableTimer.java

```
package ru.ifmo.lamport.server.co;

import java.util.TimerTask;

import ru.ifmo.lamport.server.ep.MainDialog;

import com.evelopers.unimod.core.stateworks.Event;
import com.evelopers.unimod.runtime.EventProvider;
import com.evelopers.unimod.runtime.ModelEngine;
import com.evelopers.unimod.runtime.context.StateMachineContextImpl;

/**
 * More flexible timer event provider.
 */
public class ConfigurableTimer implements EventProvider{
  /**
   * @unimod.event.descr произошло очередное срабатывание таймера
   */
  public static final String E101 = "e101";

  public static java.util.Timer t;
  private static ModelEngine engine;

  public static TimerTask makeTimerTask() {
    return new TimerTask() {
      public void run() {
        engine.getEventManager().handle(
            new Event(E101), StateMachineContextImpl.create());
      }
    };
  }

  public void init(ModelEngine engine) {
    ConfigurableTimer.engine = engine;

    TimerTask timerTask = makeTimerTask();

    t = new java.util.Timer(false);
    t.schedule(
        timerTask,
```

```
        100,
        (int) (1000 * MainDialog.instance.getTimerDelay())
      );
  }

  public void dispose() {
    t.cancel();
  }
}
```

MainDialog.java

```
package ru.ifmo.lamport.server.ep;

import java.awt.Dimension;
import java.awt.FlowLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.util.ArrayList;
import java.util.Random;

import javax.swing.BoxLayout;
import javax.swing.JButton;
import javax.swing.JCheckBox;
import javax.swing.JComboBox;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.JSlider;
import javax.swing.event.ChangeEvent;
import javax.swing.event.ChangeListener;

import ru.ifmo.lamport.algorithm.co.Process;
import ru.ifmo.lamport.algorithm.co.ServerData;
import ru.ifmo.lamport.server.AutomataManager;
import ru.ifmo.lamport.server.co.ConfigurableTimer;

/**
 * Class responsible for on-screen output data representation.
 */
public class MainDialog extends JFrame {

  /**
   * Generates random numbers according to values
   * set by sliders that control time distribution.
   */
  public static class RandomNumberGenerator {
        private Random random = new Random();
        int[] values = new int[AutomataManager.getProcessCount()];
        int s = 0;
        public int randomProcessNumber() {
          if (s == 0)
            return -1;
          int k = random.nextInt(s);
          for (int i = 0; i < values.length; i++) {
            if (k < values[i])
              return i;
            k -= values[i];
          }

          System.out.println("");
          return 0;
        }
    }
```

```
private RandomNumberGenerator randomNumberGenerator
                  = new RandomNumberGenerator();
public RandomNumberGenerator getRandomNumberGenerator() {
  return randomNumberGenerator;
}

public void updateProcessRandomWeight(int n, int value) {
  int ov = randomNumberGenerator.values[n];
  randomNumberGenerator.values[n] = value;
  randomNumberGenerator.s += (value - ov);
}

private static final long serialVersionUID = 2012762104491911559L;

  public JCheckBox boxUseTimer;

  public boolean lastVerboseDisplayValue = false;
  public JCheckBox boxVerboseDisplay;
  JButton buttonSendRandom;

public AutomataManager manager;

  private static ArrayList<ProcessPane> processPanels
                      = new ArrayList<ProcessPane>();
  public static void addProcessPane(ProcessPane p) {
    processPanels.add(p);
  }

  public void updateAllExcept(int n) {
    for (ProcessPane p : processPanels) {
    if (p.number != n)
      p.addState(null);
  }
  }

  public static MainDialog instance;

  boolean lastCheckValue = true;
public ServerControl serverControl;

private JSlider sliderTimerDelay;

/**
 * @unimod.event.descr переключение автоматического режима
 */
public static final String E1 = "e1";

public static final String[] automataTypeLabels = new String[] {
                      "Semantically constructed",
                      "Formally constructed",
                      "Optimized"
                    };

public static int getMaxWidth() {
  int m = 0;
  for (String s : automataTypeLabels)
    if (m < s.length())
      m = s.length();
  return m;
}

public int lastTimerSliderValue;
```

```
public boolean nothingHappened;


JPanel mainPanel;

JPanel controlPanel;
JPanel controlPanel2;
JPanel controlPanel3;
JPanel controlPanel1;

private JComboBox comboAutomataType;


private JPanel createHeaderPanel(int[] d) {
  JPanel headerPanel = new JPanel();


  JLabel labelAutomataType = new JLabel("Automata type", JLabel.CENTER);

  labelAutomataType.setPreferredSize(new Dimension(
        d[0],
        labelAutomataType.getPreferredSize().height
      ));
  labelAutomataType.setToolTipText("Type of automata for process");
  headerPanel.add(labelAutomataType);

  JLabel labelGivenTime = new JLabel("Given time", JLabel.CENTER);
  labelGivenTime.setPreferredSize(new Dimension(
        d[1],
        labelGivenTime.getPreferredSize().height
      ));
  labelGivenTime.setToolTipText("Relative amount of time units that " +
                  "are given to process at random");
  headerPanel.add(labelGivenTime);

  JLabel labelDoOneTurn = new JLabel("Do one turn", JLabel.CENTER);
  labelDoOneTurn.setPreferredSize(new Dimension(
        d[2],
        labelDoOneTurn.getPreferredSize().height
      ));
  labelDoOneTurn.setToolTipText("Give one time unit to process number");
  headerPanel.add(labelDoOneTurn);

  JLabel labelLastAutomataState = new JLabel("Last automata states",
                          JLabel.CENTER);
  labelLastAutomataState.setPreferredSize(new Dimension(
        d[3],
        labelLastAutomataState.getPreferredSize().height
      ));
  labelLastAutomataState.setToolTipText("Last states for automata number");
  headerPanel.add(labelLastAutomataState);

  JLabel labelRemoveProcess = new JLabel("Remove process", JLabel.CENTER);
  labelRemoveProcess.setPreferredSize(new Dimension(
        d[4],
        labelRemoveProcess.getPreferredSize().height
      ));
  labelRemoveProcess.setToolTipText("Remove process");
  headerPanel.add(labelRemoveProcess);

  return headerPanel;
}
```

```
    public void doInitialization() {

      if (mainPanel == null)
        mainPanel = new JPanel();
      else
        mainPanel.removeAll();

    mainPanel.setLayout(new BoxLayout(mainPanel, BoxLayout.PAGE_AXIS));

    if (processPanels.size() > 0) {
      mainPanel.add(createHeaderPanel(
                processPanels.get(0).getDimensions())
              );
    }
    for (ProcessPane pp : processPanels)
      mainPanel.add(pp.panel);

    if (controlPanel == null) {


      controlPanel = new JPanel();
      controlPanel.setLayout(new BoxLayout(
            controlPanel,
            BoxLayout.PAGE_AXIS
          ));

      controlPanel2 = new JPanel(new FlowLayout());
      controlPanel3 = new JPanel(new FlowLayout());
      controlPanel1 = new JPanel(new FlowLayout());

      boxVerboseDisplay = new JCheckBox();
      boxVerboseDisplay.setToolTipText("Toggle detaled output");
      boxVerboseDisplay.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e) {
          boolean b = boxVerboseDisplay.isSelected();
          if (b == lastVerboseDisplayValue)
            return;
          lastVerboseDisplayValue = b;
          for (ProcessPane p : processPanels) {
            p.stateDisplayer.clear();
          }
          repaint();
        }
      });
      controlPanel1.add(boxVerboseDisplay);
      JLabel labelVerboseDisplay = new JLabel("Verbose display");
      controlPanel1.add(labelVerboseDisplay);

      buttonSendRandom = new JButton("Random");
      buttonSendRandom.setToolTipText(
            "Give one tick to random process"
          );
      buttonSendRandom.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e) {
          MainDialog.instance.manager.sendEventToProcess(getRandomNumberGener
ator().randomProcessNumber(), Process.NEXT_TICK);
          MainDialog.instance.repaint();
        }
      });
      controlPanel1.add(buttonSendRandom);

      JButton buttonReset = new JButton("Reset");
      buttonReset.setToolTipText("Restart all processes");
      buttonReset.addActionListener(new ActionListener() {
```

```
  public void actionPerformed(ActionEvent e) {
    processPanels.clear();
    serverControl.fireEvent(ServerControl.E3);
    ServerData.reset();
  }
});
controlPanel1.add(buttonReset);


JLabel labelTimerControl = new JLabel("Timer controls: ");
controlPanel2.add(labelTimerControl);
boxUseTimer = new JCheckBox();
boxUseTimer.setToolTipText("Toggle timer usage");
boxUseTimer.addActionListener(new ActionListener() {
  public void actionPerformed(ActionEvent e) {
    boolean uts = !boxUseTimer.isSelected();
    for (ProcessPane p : processPanels) {
      p.sendButton.setEnabled(uts);
    }
    buttonSendRandom.setEnabled(uts);
    if (lastCheckValue != uts) {
      serverControl.fireEvent(ServerControl.E1);
      lastCheckValue = uts;
    }

  }
});
controlPanel2.add(boxUseTimer);
JLabel labelUseTimer = new JLabel("Use timer");
controlPanel2.add(labelUseTimer);

sliderTimerDelay = new JSlider(1, 30);
sliderTimerDelay.setToolTipText("Timer tick period");
sliderTimerDelay.addChangeListener(new ChangeListener() {
  public void stateChanged(ChangeEvent e) {
    int val = sliderTimerDelay.getValue();
    if (val == lastTimerSliderValue)
      return;
    ConfigurableTimer.t.cancel();
    ConfigurableTimer.t = new java.util.Timer(false);
    ConfigurableTimer.t.schedule(
        ConfigurableTimer.makeTimerTask(),
        100,
        (int) (1000 * MainDialog.instance.getTimerDelay())
      );
    lastTimerSliderValue = val;
  }
});
controlPanel2.add(sliderTimerDelay);
JLabel labelTimerDelay = new JLabel("Timer delay");
controlPanel2.add(labelTimerDelay);

comboAutomataType = new JComboBox(automataTypeLabels);
controlPanel3.add(comboAutomataType);
JLabel labelAutomataType = new JLabel("Select automata type");
controlPanel3.add(labelAutomataType);

JButton buttonAddAutomata = new JButton("Add process");
buttonAddAutomata.setToolTipText(
      "Add one process with automata of selected type"
    );
buttonAddAutomata.addActionListener(new ActionListener() {
  public void actionPerformed(ActionEvent e) {
    manager.addAutomata(comboAutomataType.getSelectedIndex());
```

```
            reset();
          }
        });
        controlPanel3.add(buttonAddAutomata);

        controlPanel.add(controlPanel1);
        controlPanel.add(controlPanel2);
        controlPanel.add(controlPanel3);
      }

      mainPanel.add(controlPanel);

        setContentPane(mainPanel);
      }

      MainDialog() {
        super();
        setTitle("Lamport Algorithm");

      }

    public double getTimerDelay() {
      if (sliderTimerDelay == null)
        return 1;
      else
        return Math.exp(sliderTimerDelay.getValue() / 4 - 5);
    }

    public void setLastActive(int num) {
      if (0 <= num && num < processPanels.size())
        processPanels.get(num).stateDisplayer.setLastActive();
    }

    public void reset() {
      synchronized (AutomataManager.class) {
        randomNumberGenerator = new RandomNumberGenerator();
        processPanels.clear();
        serverControl.fireEvent(ServerControl.E3);
        ServerData.reset();
      }
    }
}
```

ProcessPane.java

```
package ru.ifmo.lamport.server.ep;

import java.awt.BorderLayout;
import java.awt.Dimension;
import java.awt.FlowLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

import javax.swing.JButton;
import javax.swing.JDialog;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.JSlider;
import javax.swing.event.ChangeEvent;
import javax.swing.event.ChangeListener;

import ru.ifmo.lamport.algorithm.co.Process;
import ru.ifmo.lamport.algorithm.co.ServerData;
import ru.ifmo.lamport.server.AutomataManager;
```

```
import ru.ifmo.lamport.server.ep.StateDispalyer.ShownState;
import ru.ifmo.lamport.server.ep.StateDispalyer.State;

import com.evelopers.unimod.runtime.ControlledObject;
import com.evelopers.unimod.runtime.context.StateMachineContext;

/**
 * Represents one process on the main dialog panel.
 */
public class ProcessPane implements ControlledObject {
  int number;
  JPanel panel;
  JSlider slider;
  JButton sendButton;

  StateDispalyer stateDisplayer;

  private JButton removeButton;

  private MainDialog owner = MainDialog.instance;
  private JLabel labelType;
  private JPanel panelLabelType;
  private JPanel panelSendButton;
  private JPanel panelRemoveButton;


  public ProcessPane(int n) {

    number = n;

    panel = new JPanel(new FlowLayout(FlowLayout.CENTER));
    labelType = new JLabel();
    int automatonType = AutomataManager.automataModelList.get(n);
    String s = " " + MainDialog.automataTypeLabels[automatonType]
          + " (A" + (automatonType + 1) + ")";
    labelType.setText(s);
    panelLabelType = new JPanel(new BorderLayout());
    panelLabelType.add(labelType, BorderLayout.LINE_END);
    panelLabelType.setPreferredSize(new Dimension(
          198,
          (int) panelLabelType.getPreferredSize().getHeight())
        );
    panelLabelType.setToolTipText("Type of automata for process number "
                      + (n + 1));
    panel.add(panelLabelType);



    slider = new JSlider(0, 10);
    slider.setToolTipText("Relative amount of time units that are given " +
                "to process number " + (n + 1) + " at random");
    slider.addChangeListener(new ChangeListener() {
      public void stateChanged(ChangeEvent e) {
        MainDialog.instance.updateProcessRandomWeight(
            number,
            slider.getValue()
          );
      }
    });
    panel.add(slider);
    MainDialog.instance.updateProcessRandomWeight(number, slider.getValue());

    panelSendButton = new JPanel(new BorderLayout());
    sendButton = new JButton("Tick");
```

```
  sendButton.setToolTipText("Give one time unit to process number "
                  + (n + 1));
  sendButton.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
      MainDialog.instance.manager.sendEventToProcess(
          number,
          Process.NEXT_TICK
        );
      MainDialog.instance.repaint();
    }
  });
  panelSendButton.add(sendButton, BorderLayout.CENTER);
  panelSendButton.setPreferredSize(new Dimension(
        80,
        panelSendButton.getPreferredSize().height
      ));
  panel.add(panelSendButton);

  stateDisplayer = new StateDispalyer();
  stateDisplayer.setToolTipText("Last states for automata number "
                   + (n + 1));
  stateDisplayer.setPreferredSize(new Dimension(400, 20));
  panel.add(stateDisplayer);

  panelRemoveButton = new JPanel(new BorderLayout());
  removeButton = new JButton("Remove");
  removeButton.setToolTipText("Remove this process");
  removeButton.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
      owner.manager.removeAutomata(number);
      MainDialog.instance.reset();
    }
  });
  panelRemoveButton.add(removeButton, BorderLayout.CENTER);
  panelRemoveButton.setPreferredSize(new Dimension(
        110,
        panelRemoveButton.getPreferredSize().height
      ));

  panel.add(panelRemoveButton);
}

int[] getDimensions() {
  return new int[] {
      panelLabelType.getPreferredSize().width,
      slider.getPreferredSize().width,
      panelSendButton.getPreferredSize().width,
      stateDisplayer.getPreferredSize().width,
      panelRemoveButton.getPreferredSize().width
  };
}

void addState(ShownState s) {
  stateDisplayer.addState(s);
}

/**
 * @unimod.action.descr окончание выбора номера
 */
public void z1(StateMachineContext context) {
  owner.updateAllExcept(number);
  addState(new ShownState(
      State.CHOOSING,
      ServerData.getNumber(number),
```

```
      stateDisplayer.getStateNumber())
    );
}

void showDialog(String s) {
  JDialog d = new JDialog();
  d.setDefaultCloseOperation(JDialog.DISPOSE_ON_CLOSE);
  d.add(new JLabel(s));
  d.setModal(false);
  d.pack();
  d.setVisible(true);
}

/**
 * @unimod.action.descr вход в критическую секцию
 */
public void z2(StateMachineContext context) {
  owner.updateAllExcept(number);
  addState(new ShownState(
      State.MAIN,
      ServerData.getNumber(number),
      stateDisplayer.getStateNumber())
    );
}

/**
 * @unimod.action.descr выход из критической секции
 */
public void z3(StateMachineContext context) {
  owner.updateAllExcept(number);
  stateDisplayer.addState(new ShownState(State.FREE));
}

/**
 * @unimod.action.descr Переход в состояние 1
 */
public void z01(StateMachineContext context) {
  stateDisplayer.setStateNumber(1);
}
/**
 * @unimod.action.descr Переход в состояние 2
 */
public void z02(StateMachineContext context) {
  stateDisplayer.setStateNumber(2);
}
/**
 * @unimod.action.descr Переход в состояние 3
 */
public void z03(StateMachineContext context) {
  stateDisplayer.setStateNumber(3);
}
/**
 * @unimod.action.descr Переход в состояние 4
 */
public void z04(StateMachineContext context) {
  stateDisplayer.setStateNumber(4);
}
/**
 * @unimod.action.descr Переход в состояние 5
 */
public void z05(StateMachineContext context) {
  stateDisplayer.setStateNumber(5);
}
/**
```

```
   * @unimod.action.descr Переход в состояние 6
   */
  public void z06(StateMachineContext context) {
    stateDisplayer.setStateNumber(6);
  }
  /**
   * @unimod.action.descr Переход в состояние 7
   */
  public void z07(StateMachineContext context) {
    stateDisplayer.setStateNumber(7);
  }
  /**
   * @unimod.action.descr Переход в состояние 8
   */
  public void z08(StateMachineContext context) {
    stateDisplayer.setStateNumber(8);
  }
  /**
   * @unimod.action.descr Переход в состояние 9
   */
  public void z09(StateMachineContext context) {
    stateDisplayer.setStateNumber(9);
  }
  /**
   * @unimod.action.descr Переход в состояние 10
   */
  public void z010(StateMachineContext context) {
    stateDisplayer.setStateNumber(10);
  }
}
```

ServerControl.java

```
package ru.ifmo.lamport.server.ep;

import com.evelopers.common.exception.CommonException;
import com.evelopers.unimod.core.stateworks.Event;
import com.evelopers.unimod.runtime.EventProvider;
import com.evelopers.unimod.runtime.ModelEngine;
import com.evelopers.unimod.runtime.context.StateMachineContextImpl;


/**
 * Generates control messages for main automata.
 */
public class ServerControl implements EventProvider {

  /**
   * @unimod.event.descr изменился способ управления
   */
  public static final String E1 = "e1";

  /**
   * @unimod.event.descr инициализация
   */
  public static final String E3 = "e3";

  private ModelEngine engine;

  public void init(ModelEngine engine) throws CommonException {
    MainDialog.instance = new MainDialog();
    MainDialog.instance.serverControl = this;
    this.engine = engine;
  }
```

```
  public void fireEvent(String e) {
    engine.getEventManager().handle(
        new Event(e),
        StateMachineContextImpl.create()
      );
  }

  public void dispose() {
  }

}
```

StateDispalyer.java

```
package ru.ifmo.lamport.server.ep;

import java.awt.Color;
import java.awt.FontMetrics;
import java.awt.Graphics;
import java.awt.Image;
import java.awt.geom.Rectangle2D;
import java.util.LinkedList;

import javax.swing.JComponent;
import javax.swing.JLabel;


/**
 * Displays states of one process automata.
 */
public class StateDispalyer extends JComponent {

  enum State {
    FREE(new Color(180, 255, 180)),
    CHOOSING(new Color(255, 255, 150)),
    MAIN(new Color(255, 180, 180)),
    EMPTY(new Color(180, 255, 180)),
    VOID(Color.LIGHT_GRAY);


    Color color;
    State(Color c) {
      color = c;
    }
  }

  public static Color colorDiv2(Color c) {
    return new Color(c.getRed() * 2, c.getGreen() * 2, c.getBlue() * 2 );
  }

  public static class ShownState {
    static ShownState emptyShownState = new ShownState(State.EMPTY);
    static ShownState voidShownState = new ShownState(State.VOID);
    static {
      voidShownState.active = false;
      emptyShownState.active = false;
    }

    State state;
    int number;
    int stateNum;
    boolean active;
    public ShownState(State state, int number, int snum) {
```

```
      this.number = number;
      this.state = state;
      this.stateNum = snum;
      this.active = true;
    }
    public ShownState(State s) {
      this(s, -1, -1);
    }

    public ShownState(ShownState s) {
      this(s.state, s.number, s.stateNum);
      this.active = false;
    }
    public ShownState(ShownState p, int stateNumber) {
      this(p);
      this.stateNum = stateNumber;
    }
  }

  public void setLastActive() {
    if (states.size() > 0)
      states.getLast().active = true;
  }

  private static final long serialVersionUID = -634977408419146102 1L;

  private static final Color BORDER_COLOR = Color.BLACK;

  static int defaultMaxRectangles = 6;
  int maxRectangles;
  LinkedList<ShownState> states = new LinkedList<ShownState>();

  JLabel observer = new JLabel("Observer");

  private int stateNumber = 0;

  StateDispalyer() {
    maxRectangles = defaultMaxRectangles;
  }

  /**
   * Adds displayed state to the end of queue.
   *
   * @param s displayed state. If s is null,
   * inactive copy of last state is created.
   */
  public void addState(ShownState s) {
    if (s == null) {
      ShownState p;
      if (states.size() > 0)
        p = states.getLast();
      else
        p = ShownState.emptyShownState;
      s = new ShownState(p, stateNumber);
    }
    if (states.size() >= maxRectangles)
      states.poll();
    states.add(s);
  }

  @Override
  protected void paintComponent(Graphics g) {
    Image image = createImage(getWidth(), getHeight());
    Graphics ig = image.getGraphics();
```

```
    int cx = 0;
    int dx = (getWidth() - 2) / maxRectangles;
    FontMetrics fm = ig.getFontMetrics();
    int t = states.size();
    for (int i = 0; i < maxRectangles - t; i++)
      states.add(ShownState.voidShownState);
    for (ShownState s : states) {
      ig.setColor(s.state.color);
      ig.fillRect(cx, 0, dx + 2, getHeight() - 1);

      if (s.stateNum > -1) {
        String n = String.valueOf(s.stateNum);
        if (s.number != -1) {
          n += "(" + s.number + ")";
        }
        Rectangle2D r = fm.getStringBounds(n, g);
        ig.setColor(Color.BLACK);
        ig.drawString(
            n,
            cx + (int) Math.round((dx - r.getWidth()) / 2),
            (int) Math.round((getHeight() + 0.5 * r.getHeight()) / 2)
          );
      }

      cx += dx;
    }
    cx = 0;
    for (ShownState s : states) {
      if (s.active) {
        ig.setColor(BORDER_COLOR);
        for (int i = 0; i < 2; i++) {
          ig.drawRect(cx + i, 0 + i, dx, getHeight() - 2 - 2 * i);
        }
        ig.drawLine(cx + 1 + dx, 0, cx + 1 + dx, getHeight() - 2);
      }
      cx += dx;
    }
    for (int i = 0; i < maxRectangles - t; i++)
      states.removeLast();

    g.drawImage(image, 0, 0, observer);
  }

  public void clear() {
    if (states.size() == 0)
      return;
    ShownState s = states.getLast();
    states.clear();
    addState(s);
  }

  public void setStateNumber(int i) {
    stateNumber = i;
  }

  public int getStateNumber() {
    return stateNumber;
  }
}
```

Process.java

```
package ru.ifmo.lamport.algorithm.co;
```

```
import ru.ifmo.lamport.server.AutomataManager;

import com.evelopers.unimod.core.stateworks.Event;
import com.evelopers.unimod.runtime.ControlledObject;
import com.evelopers.unimod.runtime.EventManager;
import com.evelopers.unimod.runtime.ModelEngine;
import com.evelopers.unimod.runtime.context.StateMachineContext;
import com.evelopers.unimod.runtime.context.StateMachineContextImpl;

/**
 * Controlled object that provides corresponding automata
 * with access to data concerning one process.
 */
public class Process implements ControlledObject {

  private int i;
  private ModelEngine engine;
  private EventManager eventManager;

  public Process(int n) {
    i = n;
  }

  public void setModelEngine(ModelEngine me) {
    engine = me;
    eventManager = engine.getEventManager();
  }

  protected void handleEvent(String event) {
        eventManager.handle(new Event(event), StateMachineContextImpl.create(
));
  }

  public static final String NEXT_TICK = "e101";

  private int j = 0;

  /**
   * @unimod.action.descr число процессов
   */
  public int x1(StateMachineContext context) {
    return AutomataManager.getProcessCount();
  }

  /**
   * @unimod.action.descr choosing другого процесса
   */
  public boolean x2(StateMachineContext context) {
    //System.err.println("x2");
    return ServerData.getChoosing(j);
  }

  /**
   * @unimod.action.descr number этого процесса меньше,
   * чем number другого процесса
   */
  public boolean x3(StateMachineContext context) {
    return ServerData.getNumber(i) < ServerData.getNumber(j);
  }

  /**
   * @unimod.action.descr идентификатор другого процесса меньше
   * количества процессов
   */
```

```
public boolean x4(StateMachineContext context) {
  return j < AutomataManager.getProcessCount();
}

/**
 * @unimod.action.descr выполняется ли условие ожидания
 */
public boolean x5(StateMachineContext context) {
  return (!(ServerData.getNumber(j) == 0)) && (
      (ServerData.getNumber(j) < ServerData.getNumber(i))
      || (
          (ServerData.getNumber(j) == ServerData.getNumber(i))
          && (j < i)
        )
    );
}

/**
 * @unimod.action.descr счетчик ожидания больше 0
 */
public boolean x6(StateMachineContext context) {
  return ServerData.getWait(i) > 0;
}

/**
 * @unimod.action.descr выполнение закончилось
 */
public boolean x7(StateMachineContext context) {
  return false;
}

/**
 * @unimod.action.descr установка number текущего процесса равным
 * number другого процесса
 */
public void z11(StateMachineContext context) {
  ServerData.setNumber(i, j);
}

/**
 * @unimod.action.descr увеличение number текущего процесса на 1
 */
public void z12(StateMachineContext context) {
  ServerData.incNumber(i);
}

/**
 * @unimod.action.descr >?= для number
 */
public void z13(StateMachineContext context) {
  if (ServerData.getNumber(i) < ServerData.getNumber(j))
    ServerData.setNumber(i, j);
}

/**
 * @unimod.action.descr установка номера другого процесса в 0
 */
public void z21(StateMachineContext context) {
  j = 0;
}

/**
 * @unimod.action.descr увеличение номера следующего процесса
 */
```

```
  public void z22(StateMachineContext context) {
    j++;
  }

  /**
   * @unimod.action.descr установка choosing
   */
  public void z31(StateMachineContext context) {
    ServerData.setChoosing(i, true);
  }

  /**
   * @unimod.action.descr сброс choosing
   */
  public void z32(StateMachineContext context) {
    ServerData.setChoosing(i, false);
  }

  /**
   * @unimod.action.descr процесс простаивает
   */
  public void z4(StateMachineContext context) {

  }

  /**
   * @unimod.action.descr вход в критическую секцию
   */
  public void z51(StateMachineContext context) {
    ServerData.setWait(i, ServerData.randomInt(1, 5));
  }

  /**
   * @unimod.action.descr уменьшение счетчика ожидания
   */
  public void z52(StateMachineContext context) {
    ServerData.decWait(i);
  }

  /**
   * @unimod.action.descr выход из критической секции
   */
  public void z53(StateMachineContext context) {
    ServerData.setNumber(i, 0);
    ServerData.setWait(i, ServerData.randomInt(2, 8));
  }
}
```

ServerData.java

```
package ru.ifmo.lamport.algorithm.co;

import java.util.Random;

import ru.ifmo.lamport.server.AutomataManager;

import com.evelopers.unimod.runtime.ControlledObject;

/**
 * Storage of all data.
 */
public class ServerData implements ControlledObject {

  public static void reset() {
```

```
    choosing = new boolean[AutomataManager.getProcessCount()];
    number = new int[AutomataManager.getProcessCount()];
    wait = new int[AutomataManager.getProcessCount()];
  }

  private static boolean choosing[] =
    new boolean[AutomataManager.getProcessCount()];

  private static int number[] = new int[AutomataManager.getProcessCount()];

  private static int wait[] = new int[AutomataManager.getProcessCount()];

  public static boolean getChoosing(int i) {
    return choosing[i];
  }

  public static int getNumber(int i) {
    return (i < number.length) ? number[i] : Integer.MAX_VALUE;
  }

  public static void setNumber(int i, int j) {
    number[i] = number[j];
  }

  public static void incNumber(int i) {
    number[i]++;
  }

  public static void setChoosing(int i, boolean v) {
    choosing[i] = v;
  }

  public static void setWait(int i, int w) {
    wait[i] = w;
  }

  public static void decWait(int i) {
    wait[i]--;
  }

  public static int getWait(int i) {
    return wait[i];
  }

  private static Random random = new Random();

  public static int randomInt(int min, int max) {
    return random.nextInt(max - min + 1) + min;
  }
}
```