Опубликовано в Научно-техническом вестнике СПбГУ ИТМО. 2008. Выпуск 53. Автоматное программирование, с. 238-251.

УДК 004.4’232

НАСЛЕДОВАНИЕ АВТОМАТНЫХ КЛАССОВ С ИСПОЛЬЗОВАНИЕМ ДИНАМИЧЕСКИХ ЯЗЫКОВ ПРОГРАММИРОВАНИЯ НА ПРИМЕРЕ *RUBY*

К.И. Тимофеев, А.А. Астафуров
(Санкт-Петербургский государственный университет информационных технологий, механики и оптики)

Объектно-ориентированное программирование с явным выделением состояний совмещает в себе основные преимущества объектно-ориентированной и автоматной парадигм программирования. Основными достоинствами такого подхода являются расширяемость, гибкость и наличие механизма описания сложного поведения, реализованного на конечных автоматах. В данной работе рассматриваются объектно-ориентированный и динамический подходы к наследованию и вложению автоматных классов с использованием динамических языков программирования.
Ключевые слова: динамический язык программирования, автоматное программирование

Введение

При создании систем со сложным поведением целесообразно применять автоматное программирование, называемое также «программирование от состояний» или «программирование с явным выделением состояний». Метод разработки программного обеспечения, основанный на расширенной модели конечных автоматов, ориентирован на создание широкого класса приложений [1]. При совместном использовании объектной и автоматной парадигм программирования этот подход был назван в работе [1] «объектно-ориентированное программирование с явным выделением состояний», которое в дальнейшем будем называть «объектно-автоматное программирование». Однако при использовании автоматного программирования возникают проблемы, связанные с поддержкой автоматного кода и документации, с внесением изменений в систему, а также наглядностью и понятностью автоматного кода.

Объектно-автоматное программирование основано на объектной и автоматной парадигмах программирования. Оно совмещает в себе их основные преимущества, такие как гибкость, расширяемость и наличие мощного механизма описания сложного поведения, основанного на конечных автоматах.

Базовым понятием объектно-автоматного программирования является автоматный класс. Автоматным называется такой класс, поведение объектов которого зависит от текущего управляющего состояния. Эти классы реализуются на основе конечных автоматов [2, 3]. Наследование и вложение состояний, применяемое в объектно-автоматном программировании, позволяет значительно сократить требуемое число переходов для описания сложного автомата [4].

Однако в описываемых в работе [4] объектно-автоматных подходах не всегда удается удобно выделять условия переходов, а также писать код, который наглядно отражает переходы между состояниями, так как логика переходов обычно скрывается в методах-обработчиках входных воздействий.

Для устранения недостатков объектно-автоматного подхода, а также сохранения всех его достоинств в данной работе предлагается рассмотреть применение динамических языков программирования для построения автоматных программ. В качестве такого языка будет использован язык *Ruby*.

В данной работе будут описаны два подхода к разработке автоматных программ на языке *Ruby*, а также выполнено их сравнение с уже существующими решениями. В

качестве примера применения подходов рассмотрено создание автоматных классов для чтения и записи в файл с использованием вложения и наследования.

Обзор подходов и решений реализации автоматов

Существует несколько подходов к реализации автоматов, каждый из которых имеет свои достоинства и недостатки. Рассмотрим их.

1. Полностью ручное программирование. Существует различные методы реализации этого подхода, например, использование одного или нескольких операторов `switch`, определяющих действия программы в зависимости от текущего состояния [5], и использование шаблона проектирования *State* [6]. Несмотря на высокую производительность и полный контроль над получаемым кодом, эти методы обладают недостатками: низкая читаемость кода из-за применения множества вложенных операторов `switch` (для первого похода) и большая трудоемкость его написания (для второго подхода).
2. Автоматическая генерация кода по диаграмме переходов. Обычно при таком подходе генерируется код, аналогичный тому, который можно получить с использованием ручного программирования. Эта группа методов обладает следующими достоинствами: высокая скорость создания программного кода и возможность его создания экспертами предметной области, которые не имеют опыта использования языков программирования высокого уровня. Недостатками этого подхода являются: низкая читаемость, связанная с тем, что в качестве целевого языка используется императивный язык, например, *Java*, который не способен полностью отразить декларативную сущность диаграммы переходов автомата [7], потеря информации, специфичной для логики диаграмм переходов (вершины диаграммы и ее переходы заменяются их реализациями на целевом языке – классами и кодом выполнения переходов), недостаточная степень контроля над получаемым кодом и невозможность его ручного изменения.
3. Ручное написание кода с применением специальной библиотеки. В этом случае происходит перенос диаграммы переходов в вызовы указанной библиотеки, которая по этим инструкциям строит внутреннее представление рассматриваемой диаграммы. Затем по этому представлению происходит реализация автомата. Основным достоинством этого подхода является то, что вызовы библиотеки отражают семантику диаграммы переходов (каждый вызов может, например, соответствовать объявлению состояния или перехода). Это позволяет создавать читаемый код, который легко поддерживать [8].

В приведенных ниже работах используется третий подход к реализации автоматных классов, однако каждое из указанных решений наряду с достоинствами, обладает и недостатками.

1. В работе [8] используется динамический язык программирования *Ruby*, с помощью которого была разработана библиотека `STROBE`, что позволило удобно преобразовать автомат из графической нотации в программный код. Однако в работе [8] не было реализовано наследование автоматных классов.
2. В работе [7] применяется декларативный подход к реализации автоматных объектов при помощи объектно-ориентированных императивных языков программирования со статический проверкой типов. Используя язык программирования *C#*, было реализовано наследование и вложение автоматов. В то же время код программы обладает синтаксической избыточностью, что не позволяет легко модифицировать логику переходов.
3. Плагин *Acts as State Machine* [9] для *Ruby on Rails* позволяет описывать логику веб-приложения с использованием автоматного программирования. *Ruby on Rails* – это фреймворк для разработки веб-приложений. Плагин – это компонент, который мож-

но добавить в приложение *Ruby on Rails,* расширив его функциональность [10]. Этот плагин обладает простым синтаксисом. Однако в нем не было реализовано наследование и вложение автоматных классов.

В данной работе устранены такие недостатки приведенных выше решений, как синтаксическая избыточность, невозможность наследования или вложения автоматов. Для этого реализованы два подхода к преобразованию диаграмм переходов в программный код. Первый подход будет использовать объектно-ориентированные свойства языка *Ruby*, второй – динамические.

В качестве примера будут реализованы автоматные классы для чтения из файла и записи в файл, поведение которых может быть обобщено и структурировано с помощью наследования. Эти классы образуют иерархию, показанную на рис. 1.

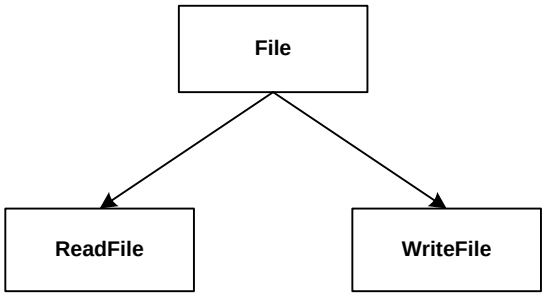


Рис. 1. Иерархия классов доступа к файлу

Динамические языки программирования

Языки этого класса характеризуется динамической типизаций и возможностью выполнения кода, созданного в процессе работы программы. Динамические языки программирования позволяют:

- расширять программу, классы или объекты с помощью добавления нового кода во время выполнения программы;
- выполнять любые инструкции в виде текста (метод `eval`);
- использовать:
  - лямбда-функции, которые обозначают безымянные функции;
  - замыкания – способность лямбда-функции запоминать свой контекст выполнения;
  - функции высшего порядка – возможность передать функцию в качестве аргумента и возможность вернуть функцию в качестве результата работы;
  - отражения – анализ типов и метаданных, представление полного текста программы как данных;
  - макросы – правило или шаблон для преобразования входной последовательности в соответствии с объявленной процедурой. Результатом выполнения макросов является программа исходного языка – макросы расширяют исходный язык.

Стоит отметить, что неотъемлемой частью функционального программирования являются функции высшего порядка и замыкания, что позволяет в некоторой степени объединить функциональный и динамические подходы.

Функциональные языки программирования

Близость к математической формализации и функциональная ориентированность дают следующие преимущества этому классу языков программирования:

- сложные программы строятся на основе агрегирования функций;

- полиморфизм (отличается по сути от аналогичного термина в объектно-ориентированном программировании) – обеспечение возможности обработки разнородных данных [11];
- простота тестирования и верификации на основе строгого математического доказательства корректности программ;
- унификация представления программы и данных;

Таким образом, при создании программ на функциональных языках программист основное внимание обращает на предметную область и в меньшей степени заботится о рутинных операциях («сборка мусора», оптимизация и т.д.).Элементы функционального программирования используются и в императивных языках, например, в языке C# 3.0 существуют лямбда-функции, функции высшего порядка и замыкания.

Особенности языка *Ruby*

Ruby – кросс-платформенный, объектно-ориентированный динамический язык программирования с элементами функционального стиля [12]. Он имеет все черты объектно-ориентированных языков, такие как инкапсуляция, наследование и полиморфизм. Множественное наследование реализуется с помощью *технологии примесей*. Примесь – это механизм расширения одного класса методами другого. Язык *Ruby* содержит в себе все элементы динамических языков программирования, которые были указаны выше, и частично поддерживает функциональное программирование за счет лямбда-функций, замыканий и функций высшего порядка.

Динамические и функциональные свойства языка *Ruby* позволяют:

1. Получить автомат, код которого удобен для чтения и дальнейшего сопровождения, благодаря возможности разработки предметно-ориентированного языка на макросах рассматриваемого языка.

Предметно-ориентированный язык (*Domain Specific Language, DSL*) – это язык программирования, который разработан специально для решения определенного круга задач [13]. Преимуществами применения *DSL* являются:

   - возможность создания решения в терминах предметной области. В результате эксперты данной предметной области могут понимать, проверять и модифицировать написанный код;
   - самодокументированный код;
   - повышение качества, надежности и сопровождаемости программного обеспечения.

2. Легко верифицировать автомат. Для этого потребуется решить две задачи:
   - доказать корректность построения структуры автомата с помощью DSL. Так как свойством функциональных языков программирования является отсутствие побочных эффектов (побочным эффектом считается изменение функцией состояния своего окружения), то программа будет состоять из набора несвязанных друг с другом функций, что связано с тем, что результат выполнения одной функции не зависит от результата выполнения другой. Таким образом, для верификации DSL будет достаточно убедиться в корректной работе каждой отдельной функции;
   - верифицировать автомат, применяя темпоральную логику.
3. Решить вопрос изоморфного переноса графической нотации в программный код [14] – при реализации группового перехода не требуется дублировать код перехода для вложенных состояний.

Использование графической нотации для описания логики автоматных классов

При проектировании автоматных классов предлагается использовать *диаграммы поведения*, являющиеся расширенной версией графов переходов. Отличительной особенностью диаграмм поведения, представленных в работе [4], является возможность описания декомпозиции и структурирования логики автоматных классов с помощью наследования.

При наследовании производный автоматный класс может расширять и модифицировать поведение базовых автоматных классов. Такая модификация базовых классов основана на переопределении их состояний. В производный класс могут быть добавлены новые состояния и переходы между ними [4].

Диаграмма поведения обладает следующими свойствами.

1. Диаграмма изображает одно или несколько состояний системы и переходы между ними.
2. Состояния на диаграмме могут быть объединены в группы, которые могут быть вложены друг в друга.
3. Переходы могут начинаться и заканчиваться в состоянии или в группе состояний (переходы, начинающиеся в группе состояний, называются групповыми переходами).
4. Переходы могут начинаться и заканчиваться в одном и том же состоянии. При этом они называются петлями.

Основные элементы графической нотации [4] представлены на рис. 2.

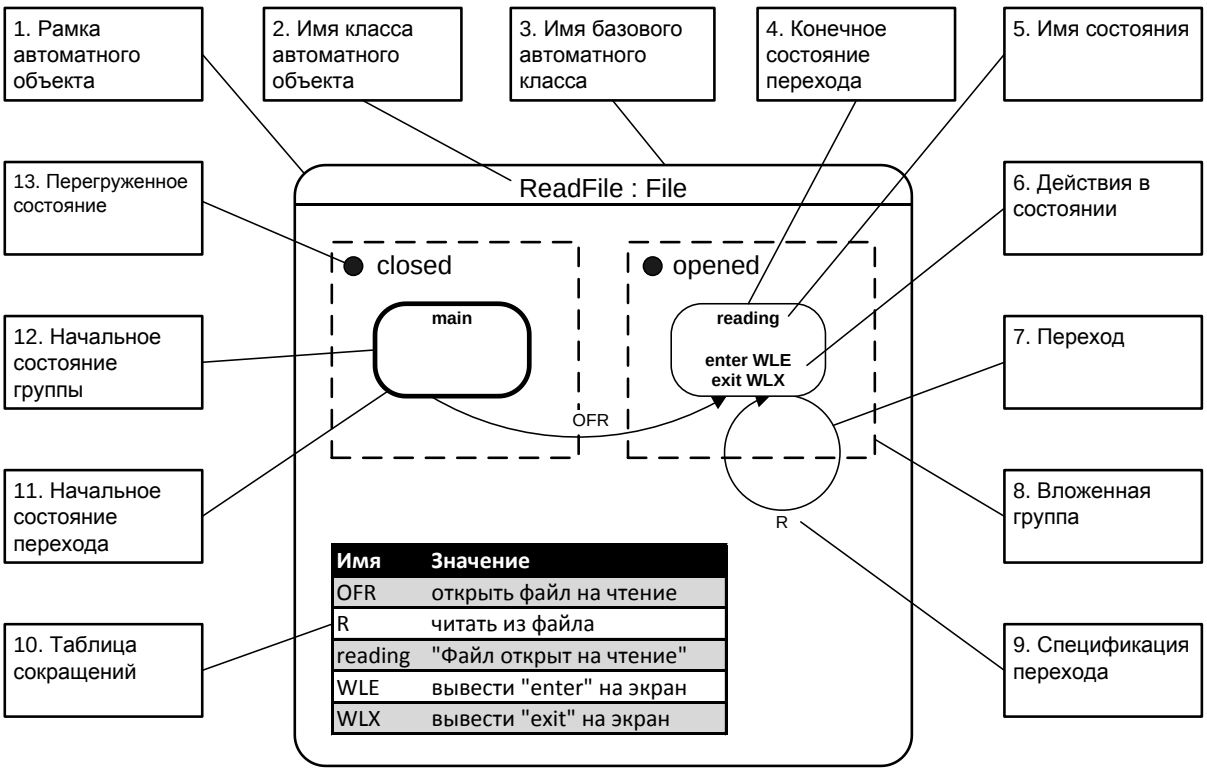


Рис 2. Основные элементы графической нотации

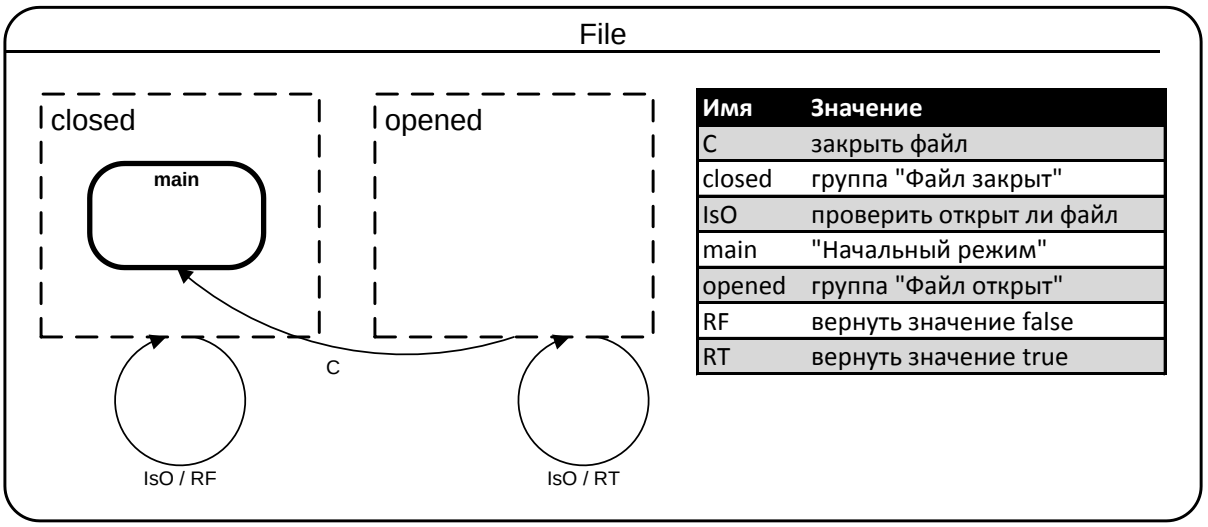
Графическая нотация используется в данной работе для описания автоматных классов.

Наследование и вложение автоматов с использованием объектно-ориентированных особенностей языка *Ruby*

На рис 3. представлен абстрактный автоматный класс для работы с файлом (`File`), который состоит из двух вложенных групп `closed` и `opened`. Вложенная группа `closed` включает состояние `main`. Вложенная группа `opened` не содержит ни одного состояния и будет использована при создании наследуемых автоматов `ReadFile` и `WriteFile` (рис. 4).

Определены общие для всех файлов переходы:

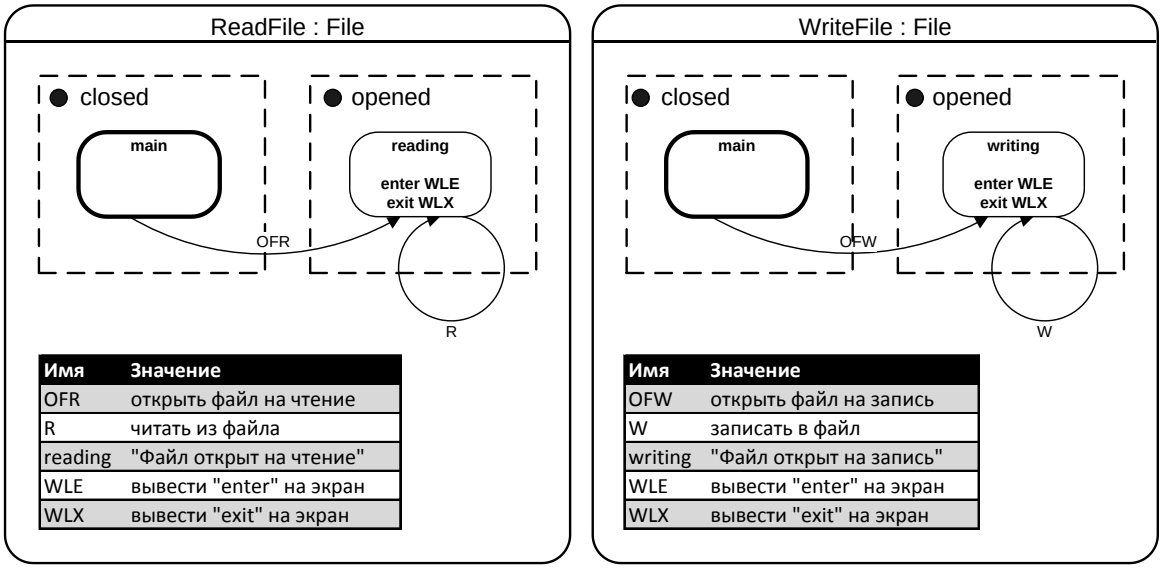
- `IsO` – позволяет определить, открыт ли в данный момент файл;
- `C` – закрывает файл.



Имя	Значение
C	закрыть файл
closed	группа "Файл закрыт"
IsO	проверить открыт ли файл
main	"Начальный режим"
opened	группа "Файл открыт"
RF	вернуть значение false
RT	вернуть значение true

Рис 3. Абстрактный класс работы с файлом

Диаграмма поведения классов `ReadFile` и `WriteFile`, построенная с использованием наследования, приведена на рис. 4. Корневым элементом предлагаемой иерархии является абстрактный класс `File`, обобщающий некоторые аспекты доступа к файлу.



Имя	Значение
OFR	открыть файл на чтение
R	читать из файла
reading	"Файл открыт на чтение"
WLE	вывести "enter" на экран
WLX	вывести "exit" на экран

Имя	Значение
OFW	открыть файл на запись
W	записать в файл
writing	"Файл открыт на запись"
WLE	вывести "enter" на экран
WLX	вывести "exit" на экран

Рис 4. Реализуемые классы ReadFile и WriteFile

Автомат `ReadFile` наследует от автомата `File` группы `closed` (содержит состояние `main`) и `opened` (не содержит ни одного состояния). В группу `opened` добавляется новое состояние `Reading` с действием `WLE` при входе в состояние и действием `WLX` при выходе из состояния. Добавляются новые переходы `OFR` (из состояния `Main` в состояние `Reading`) и `R` (петля в состоянии `Reading`).

Автомат `WriteFile` наследует от автомата `File` группы `closed` (содержит состояние `main`) и `opened` (не содержит ни одного состояния). В группу `opened` добавляется новое состояние `Writing` с действием `WLE` при входе в состояние и действием `WLX` при выходе из состояния. Добавляются новые переходы `OFW` (из состояния `Main` в состояние `Writing`) и `W` (петля в состоянии `Writing`).

Реализуем приведенные выше автоматные классы, используя объектно-ориентированные особенности языка *Ruby*. При этом каждый автоматный класс и каждое состояние являются отдельным классом.

Разработаем абстрактный автоматный класс `File`. Для этого предварительно реализуем все его состояния и группы состояний. Приведенный ниже класс отвечает за создание состояния `main`, которое наследуется от класса абстрактного состояния `State`.

Созданное состояние имеет имя `main`:

```
class Main < State
 def initialize container
 super :Main, container
 end
end
```

Разработаем группу `opened`. Она имеет переход `C` в состояние `main` с действием на этом переходе `RT` (рис. 3). Переходы задаются публичными методами (например, переход `C`), тогда как действия задаются приватными методами (например, действие `RT`):

```
class Opened < Automaton
 def initialize name, container=nil
 super name, container
 end

 def C
 @container.state :Closed
```

```
 RT()
 end

 private
 def RT; true; end
end
```

Аналогичным образом создадим вложенную группу `closed`:

```
class Closed < Automaton
 def initialize name, container=nil
 super name, container
 automaton Main.new(self)
 initial :Main
 end

 def IsO
 @container.state :Main
 RF()
 end

 private
 def RF; false; end
end
```

После того, как были созданы все состояния и вложенные группы, можно создать автомат `File`, который будет включать в себя две вложенные группы `opened` и `closed`.

```
class FileAbstract < Automaton
 def initialize name, container=nil
 super name, container
 automaton Closed.new(:Closed, self), Opened.new(:Opened, self)
 initial :Closed
 end
end
```

После того, как был описан абстрактный автоматный класс для работы с файлами (`FileAbstract`), создадим наследуемый от него автомат `ReadFile` (рис. 4). Для этого создадим новое состояние `Reading`:

```
class Reading < State
 def initialize container
 super :Reading, container
 end

 def R; @container.state :Reading; end
end
```

Группа `ReadOpened` наследована от группы `opened`. Следовательно, она будет иметь все переходы и состояния, которые были объявлены в группе `opened`. Однако в эту группу требуется добавить состояние `Reading`, что достигается вызовом метода `automaton`. Этот метод используется для создания вложенной группы с заданными состояниями. Начальным состоянием вложенной группы `ReadOpened` является состояние `Reading`:

```
class ReadOpened < Opened
 def initialize name, container=nil
 super name, container
 automaton Reading.new(self)
 initial :Reading
 end
end
```

Так же создадим оставшиеся состояния `ReadMain`, `ReadFile` и группу `ReadClosed`:

```
class ReadMain < Main
 def OFR; @container.state :Reading; end
end

class ReadClosed < Closed
 def initialize name, container=nil
 super name, container
 automaton ReadMain.new(self)
 end
end

class ReadFile < FileAbstract
 def initialize name, container=nil
 super name, container
 automaton ReadOpened.new(:Opened, self),
         ReadClosed.new(:Closed, self)
 end
end
```

В результате создан программный код, отображающий автоматы, представленные на рис. 3, 4.

Отметим следующее достоинство использования объектно-ориентированного подхода для реализации наследования и вложения автоматов [7]: каждое состояние и каждый автомат является отдельным классом, что позволяет сохранить иерархию автомата при переносе его в объектно-ориентированный код. Недостаток этого подхода – *синтаксическая избыточность*, связанная с необходимостью использования синтаксических конструкций языка, что не позволяет легко модифицировать логику переходов и расширять код.

В работе [4] графическая нотация может быть преобразована в программный код лишь одним способом: с помощью объектно-ориентированного программирования, тогда как динамические языки программирования позволяют применить еще и другой способ, основанный на метапрограммировании, который будет рассмотрено ниже.

Наследование и вложение автоматов с помощью динамических свойств языка *Ruby*

Этот подход устраняет синтаксическую избыточность, которая была отмечена в качестве недостатка предыдущего решения.

В результате применения динамического языка *Ruby* был разработан специальный предметно-ориентированный язык, который состоит из двух основных конструкций:

- `current` – начальное состояние автомата;
- `state` – создает новое состояние, которое может иметь несколько *ключевых параметров* (ключевые параметры позволяют передавать аргументы функции по имени параметра) :
  - `:enter` – действие при входе в состояние;
  - `:exit` – действие при выходе из состояния.
- `transition` – переход из одного состояния в другое. Имеет несколько ключевых параметров:
  - `:from` – состояние, из которого происходит переход;
  - `:to` – состояние, в которое происходит переход;
  - `:guard` – условие, при котором может произойти переход;
  - `:event` – действие на переходе.

Ключевые параметры `:guard`, `:exit` и `:enter` являются лямбда-функциями. Условие, при котором выполняется переход, является лямбда-предикатом – лямбда-функцией, которая возвращает значение «истина» или «ложь».

Приведем код, который реализует автоматы, изображенные на рис. 3, 4:

```
class AbstractFile < Automaton::Base
 class << self
 def RT; lambda { true } end
 def RF; lambda { false } end
 def IsO; lambda { @file.is_opened? } end
 end

 current :main

 state :closed, :enter => lambda { { :current_state => :main }
         }
 state :opened
 state :main

 transition :from => :opened, :to => :main,
 :event => :C

 transition :from => :closed, :to => :closed,
 :event => :IsO,
 :proc => RF()

 transition :from => :opened, :to => :opened,
 :event => :IsO,
 :proc => RT()
end

class ReadFile < AbstractFile
 class << self
 def WLE; lambda { puts "enter" } end
 def WLX; lambda { puts "exit" } end
 def R; lambda { @file.read } end
 def OFR; lambda { |name| @file.open(name, "r") } end
 end

 state :reading,
 :enter => WLE(),
 :exit => WLX()

 state :opened, :enter => lambda { { :current_state => :reading
         } }

 transition :from => :reading, :to => :reading,
 :event => :R

 transition :from => :main, :to => :reading,
 :event => :OFR
end

class WriteFile < AbstractFile
 class << self
 def WLE; lambda { puts "enter" } end
 def WLX; lambda { puts "exit" } end
 def R; lambda { @file.write } end
 def OFW; lambda { |name| @file.open(name, "w") } end
 end
```

```
 state :writing,
 :enter => WLE(),
 :exit => WLX()

 state :opened, :enter => lambda { { :current_state => :writing
         } }

 transition :from => :writing, :to => :writing,
 :event => :W

 transition :from => :main, :to => :writing,
 :event => :OFW
end
```

Рассмотрим конструкции предметно-ориентированного языка, использованные в этой программе:

- установить начальное состояние автомата в `main`;

```
current :main
```

- создать новое состояние с именем `writing`. При входе в это состояние выполняется функция `WLE()`, а при выходе из него – функция `WLX()`.

```
state :writing,
 :enter => WLE(),
 :exit => WLX()
```

- создать переход из состояния `closed` в состояние `closed`, что образует петлю при входном воздействии `IsO()`. В случае перехода будет выполнена функция `RF()`:

```
transition :from => :closed, :to => :closed,
 :event => :IsO,
 :proc => RF()
```

Рассмотрим наследование автомата `ReadFile`. Для этого необходимо добавить новое состояние `Reading` и два новых перехода:

```
class ReadFile < AbstractFile
 state :reading,
 :enter => WLE(),
 :exit => WLX()

 transition :from => :reading, :to => :reading,
 :event => :R

 transition :from => :main, :to => :reading,
 :event => :OFR
end

Дополнительно потребуется определить методы R, WLE, WLX и OFR:
class ReadFile < AbstractFile
 class << self
 def WLE; lambda { puts "enter" } end
 def WLX; lambda { puts "exit" } end
 def R; lambda { @file.read } end
 def OFR; lambda { |name| @file.open(name, "r") } end
 end
end
```

Таким образом, преимуществом данного подхода является отсутствие синтаксической избыточности, которая позволяет использовать такие достоинства *DSL*, как самодокументированный код, простота его понимания и расширяемость. В то же время существенным недостатком подхода является потеря иерархии родительского автомата: состояние автомата не является отдельным классом, как было при использовании объектно-ориентированного подхода.

Протоколирование

Благодаря такому свойству динамических языков программирования, как макросы, можно разработать модуль, который будет автоматически отслеживать переходы, и выводить об этом соответствующую информацию. При этом не важно, был ли использован объектно-ориентированный или динамический подход к реализации автоматов.

Рассмотрим пример для автомата `ReadFile`:

```
class ReadFile < AbstractFile
 def initialize name, container=nil
 super name, container
 automaton ReadOpened.new(:Opened, self),
         ReadClosed.new(:Closed, self)
 end

 tracer
end
```

Этот класс отличается от использованного выше лишь одной строкой `tracer`, которая является макровызовом и создает для каждого перехода и состояния дополнительный метод, который выводит отладочную информацию. Она состоит из названия метода, который был вызван, и переданных ему аргументов. Дополнительно можно вывести информацию о времени выполнения того или иного метода:

```
ReadMain::OFR begin
In the OFR
ReadMain::OFR end
```

Заключение

В настоящей работе были разработаны два подхода, которые позволяют изоморфно переносить диаграммы состояний в диаграммы поведения, а после этого изоморфно преобразовывать полученный результат в программный код. При этом были рассмотрены объектно-ориентированные и динамические языки свойства языка программирования *Ruby*.

Достоинством использования объектно-ориентированного подхода к реализации наследования и вложения автоматов [4] является то, что каждое состояние и каждый автомат – это отдельный класс, что позволяет сохранить иерархию автомата при переносе его в объектно-ориентированный код. Однако данный подход обладает таким недостатком, как синтаксическая избыточность, что связано с необходимостью использования синтаксических конструкций языка и не позволяет легко модифицировать и расширять код.

Показано, что для динамических языков программирования может быть разработан предметно-ориентированный язык (*DSL*), который позволяет:

- создать решение в терминах предметной области. В результате эксперты в этой области могут понимать, проверять и модифицировать написанный код;
- обеспечить самодокументированный код.

Недостатком подхода является потеря иерархии родительского автомата, так как состояние автомата не является отдельным классом, как было обеспечено в объектно-ориентированном подходе.

Таким образом, оптимальным решением будет являться объединение объектно-ориентированного и динамического подходов: предметно-ориентированный язык, разработанный с использованием динамического языка программирования *Ruby*, будет использован для создания классов при применении объектно-ориентированного подхода.

В данной статье не был рассмотрен чистый функциональный подход к реализации конечных автоматов. Этот подход позволит более эффективно верифицировать автоматы. Кроме этого, в дальнейшем планируется расширить наследование автоматных классов до множественного наследования.

Литература

1. Поликарпова Н.И., Шалыто А.А. Автоматное программирование. Учебно-методическое пособие. – СПбГУ ИТМО, 2007. – Режим доступа: http://is.ifmo.ru/books/_umk.pdf.
2. Шалыто А.А., Туккель Н. И. SWITCH-технология – автоматный подход к созданию программного обеспечения «реактивных» систем. // Программирование. – 2001. – № 5. – С.45–62. – Режим доступа: http://is.ifmo.ru/works/switch/1/.
3. Шалыто А.А., Туккель Н.И. От тьюрингова программирования к автоматному // Мир ПК. – 2002. – № 2. – С. 144–149. – Режим доступа: http://is.ifmo.ru/works/turing/.
4. Шопырин Д.Г., Шалыто А.А. Графическая нотация наследования автоматных классов // Программирование. – 2007. – № 5. – С. 62–74. – Режим доступа: http://is.ifmo.ru/works/_12_12_2007_shopyrin.pdf.
5. Шалыто А.А., Туккель Н.И. Реализация автоматов при программировании событийных систем // Программист. – 2002. – № 4. – С.74–80. – Режим доступа: http://is.ifmo.ru/works/evsys/.
6. Гамма Э., Хелм Р., Джонсон Р., Влиссидес Дж. Приемы объектно-ориентированного проектирования. Паттерны проектирования. – СПб.: Питер, 2007.
7. Астафуров А.А., Шалыто А.А. Декларативный подход к вложению и наследованию автоматных классов при использовании императивных языков программирования. // Software Conference (Russia). – М.: ТЕКАМА. 2007. – С. 230–238. – Режим доступа: http://is.ifmo.ru/works/_astafurov_secr_word_2003.pdf.
8. Степанов О.Г., Шалыто А.А. Предметно-ориентированный язык автоматного программирования на базе динамического языка Ruby // Информационно-управляющие системы. – 2007. – № 4. – С. 22–27. – Режим доступа: http://is.ifmo.ru/works/_2007_10_05_aut_lang.pdf.
9. Scott B. Acts As State Machine. – Режим доступа: http://agilewebdevelopment.com/plugins/acts_as_state_machine.
10. Obie F. The Rails Way. – NJ: Addison-Wesley, 2007
11. Field A., Harrison P. Functional Programming. – Wokingham: Addison-Wesley, 1993
12. Thomas D., Fowler C., Hunt A. Programming Ruby. – Texas: Pragmatic Bookshelf, 2004.
13. Parr T. The Definitive Antlr Reference: Building Domain-Specific Languages. – Texas: Pragmatic Bookshelf, 2007.
14. Заякин Е.А., Шалыто А.А. Метод устранения повторных фрагментов кода при реализации конечных автоматов. – СПбГУ ИТМО, 2007. – Режим доступа: http://is.ifmo.ru/projects/life_app/