Automata Classes Inheritance in Dynamic Language Ruby

Artyom Astafurov (DataArt), Kirill Timofeev (DataArt), Anatoly Shalyto (SPbSUITMO)

Abstract (English)

Automata-based programming is often used for creating systems with the complex behavior. However the automata-based programming has problems such as support of automata-based code and documentation, further system improvements and code clarity. Object-automata approach is based on the object-oriented and the automata-based approaches and combines their main advantages, such as a flexibility, scalability and availability of a powerful mechanism for describing complex systems based on the finite state machines. This approach allows to resolve the problems described above.

In existing object-automata approaches it is hard to encapsulate the transition clauses and develop code that clearly reflects the transitions between states as the transition logic is often hidden in the handler methods of incoming actions. In order to resolve the problems of existing object-automata approaches and retain all their benefits it is suggested to use dynamic programming languages for creating automata programs. This paper describes two approaches based on object-automata approach that use object-oriented and dynamic Ruby properties.

This paper shows that using object-oriented features of Ruby each state and automata can be represented as a separate class. It helps to maintain the automata hierarchy in the object-oriented code. However this approach has disadvantages, such as syntax redundancy that complicates code modifications and improvements.

With the help of Ruby's dynamic language features a domain-specific language (DSL) can be developed. The DSL will allow to an isomorphic translation of state charts to code and solve problems of pure object-oriented approach. However this approach has a disadvantage of losing the automata hierarchy because each state and automata are not represented as a separate class.

***Keywords**: Automata-based programming; object-oriented programming; object-automata approach; functional programming; dynamic languages; domain-specific languages; DSL; Ruby.*

Наследование автоматных классов с использованием динамических языков программирования на примере Ruby

Артем Астафуров (DataArt), Кирилл Тимофеев (DataArt), Анатолий Шалыто (SPbSUITMO)

Abstract (Russian)

При создании систем со сложным поведением целесообразно применять автоматное программирование. Однако при его использовании 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: гибкость, расширяемость и наличие мощного механизма описания сложного поведения, основанного на конечных автоматах, – что позволяет решить описанные выше проблемы.

Однако в существующих объектно-автоматных подходах не всегда удается удобно выделять условия переходов и 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, рассматривается применение динамических языков программирования для построения автоматных программ. В данной работе описаны два подхода, использующие объектно-ориентированные и динамические свойства языка Ruby.

Показано, что достоинством использования объектно-ориентированных свойств языка Ruby является то, что каждое состояние и автомат представлены отдельным классом. Это позволяет сохранить иерархию автомата при переносе его в объектно-ориентированный код. Однако этот подход обладает таким недостатком, как синтаксическая избыточность, что иногда может затруднять модификацию и расширение кода.

При помощи динамических свойств языка Ruby может быть разработан предметно-ориентированный язык (DSL), который позволяет изоморфно переносить диаграммы состояния в программный код, и решает проблемы предыдущего подхода. Однако недостатком динамического подхода является потеря иерархии родительского автомата при наследовании, так как состояние и автомат не представлены отдельными классами.

***Keywords**: Автоматное программирование; объектно-ориентированное программирование; объектно-ориентированное программирование с явным выделением состояний; функциональное программирование; динамические языки программирования; предметно-ориентированный язык; DSL; Ruby.*

1. Введение

При создании систем со сложным поведением целесообразно применять автоматное программирование называемое также «программирование от состояний» или «программирование с явным выделением состояний» [1]. 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.

При совместном использовании объектной и автоматной парадигм программирования этот подход был назван в работе [1] «объектно-ориентированным программированием с явным выделением состояний», которое в дальнейшем будем называть «объектно-автоматное программирование».

Объектно-автоматное программирование совмещает в себе основные преимущества объектной и автоматной парадигм, такие как гибкость, расширяемость и наличие мощного механизма описания сложного поведения, основанного на конечных автоматах, – что позволяет решить описанные выше проблемы.

Наследование и вложение состояний, применяемое в объектно-автоматном программировании, позволяет значительно сократить требуемое число переходов для описания сложного автомата [2].

Однако в описываемых в этой статье 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 этих 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 предлагаемых подходов рассмотрено создание автоматных классов для чтения и записи в файл с использованием вложения и наследования.

2. Обзор подходов и решений реализации автоматов

Существует несколько подходов к реализации автоматов: полностью ручное программирование, автоматическая генерация кода по диаграмме переходов и ручное написание кода с применением специальной библиотеки [3].

В приведённых ниже работах используется ручное написание кода с применением специальной библиотеки для реализации автоматных классов, однако каждое из указанных решений наряду с достоинствами, обладает и недостатками.

1. В работе [3] используется динамический язык программирования *Ruby*, с помощью которого была разработана библиотека `STROBE`, что позволило удобно отобразить автомат из графической нотации в программный код. Однако в этой работе не было реализовано наследование автоматных классов.

2. В работе [4] 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 может затруднять модификацию логики переходов.

3. Плагин *Acts as State Machine* [5] для *Ruby on Rails* позволяет описать логику веб-приложения с использованием автоматной парадигмы программирования. Этот плагин обладает простым синтаксисом, однако в нём не было реализовано наследование и вложение автоматных классов.

В данной работе устраняются недостатки приведённых выше решений. Для этого реализованы два подхода к переносу *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.

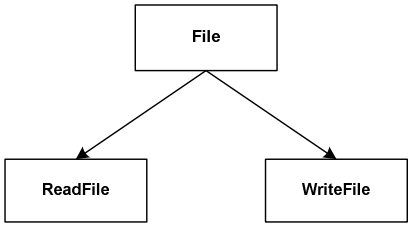


Рис. 1. Иерархия классов доступа к файлу

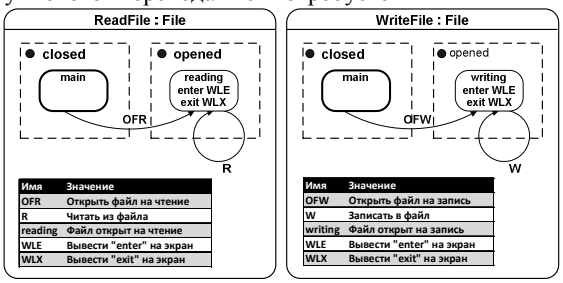
3. Особенности языка Ruby

Ruby – кросс-платформенный, объектно-ориентированный динамический язык программирования с элементами функционального стиля [6]. Эти свойства языка *Ruby* позволяют:

1. Получить автомат, код которого удобен для чтения и дальнейшего сопровождения, благодаря возможности разработки предметно-ориентированного языка (*DSL*). Преимуществами применения *DSL* являются: возможность создания решения в терминах предметной области, таким образом, эксперты данной предметной области могут понимать, проверять и модифицировать написанный код; самодокументированный код; повышение качества, надёжности и сопровождаемости программного обеспечения.

2. Легко верифицировать автомат. Для этого потребуется решить две задачи: доказать корректность построения структуры автомата с помощью *DSL* и верифицировать автомат, применяя темпоральную логику. Так как свойством функциональных языков программирования является отсутствие побочных эффектов, то для доказательства корректности построения структуры автомата с помощью *DSL* будет достаточно убедиться в корректной работе каждой отдельной функции.

3. Решить вопрос изоморфного переноса графической нотации в программный код [8] – при реализации группового перехода не потребуется дублирование кода перехода для вложенных состояний.



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

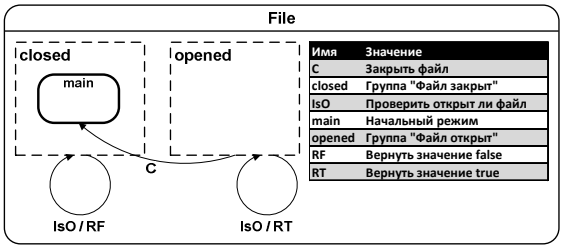


Рис. 2. Абстрактный класс работы с файлом

Используя графическую нотацию [2], на рис. 2

Рис. 3. Реализуемые классы `ReadFile` и `WriteFile`

представлен абстрактный автоматный класс для работы с файлом (`File`) Определены общие для всех файлов переходы:

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

Диаграмма поведения классов `ReadFile` и `WriteFile`, построенная с использованием наследования, приведена на рис. 3. Корневым элементом предлагаемой иерархии является абстрактный класс `File`, обобщающий некоторые аспекты доступа к файлу.

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 при входе WLE и действием при выходе WLX. Добавляются новые переходы OFR (из состояния Main в состояние Reading) и R (петля для состояния Reading). Аналогичным образом строится автомат WriteFile наследованный от автомата File.

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

Разработаем абстрактный автоматный класс File. Для этого предварительно реализуем все его состояния и вложенные группы состояний. Приведённый 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 при переходе RT (рис. 2). 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. После чего создадим автомат File, который будет включать в себя разработанные выше вложенные группы Opened и Closed.

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

Далее создадим автомат ReadFile (рис. 3) наследуемый от автомата File, для этого разработаем 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 – данный метод используется для создания вложенной группы с заданными состояниями.

Начальным состоянием вложенной группы ReadOpened будет состояние Reading, что достигается вызовом метода initial.

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

Аналогичным образом создадим состояние ReadMain, группу ReadClosed и автоматы ReadFile и WriteFile (рис. 3). В результате был создан программный код, отображающий автоматы, представленные на рис. 2, 3.

Можно отметить следующее достоинство использования объектно-ориентированного подхода к реализации наследования и вложения автоматов [4]: каждое состояние и автомат является отдельным классом, что позволяет сохранить иерархию автомата при переносе его в объектно-ориентированный код.

Недостаток данного подхода: синтаксическая избыточность, что не позволяет легко модифицировать логику переходов и расширять код.

В работе [2] графическая нотация может быть отображена в программный код лишь одним способом: с помощью объектно-ориентированного программирования, тогда как динамические языки программирования позволяют применить ещё и второй способ, основанный на метапрограммировании, который будет рассмотрен ниже.

5. 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 трёх основных конструкций:

`current` – устанавливает начальное состояние автомата.

`state` – создаёт новое состояние, которое может иметь несколько ключевых параметров:

- `:enter` – действие при входе в состояние;
- `:exit` – действие при выходе из состояния.

`transition` – создаёт переход из одного состояния в другое, который может иметь следующие ключевые параметры:

- `:from` – состояние, из которого происходит переход;
- `:to` – состояние, в которое происходит переход;
- `:guard` – условие, при котором произойдёт переход;
- `:event` – название перехода;
- `:proc` – действие на переходе.

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

Приведем код, который реализует автоматы, изображённые на рис. 2, 3:

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

  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
```

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 данное состояние выполняется функция `WLE()`. При выходе из данного состояния выполняется 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`, которое будет являться основным для группы `Opened`, и два новых перехода `R` и `OFR`:

```
class ReadFile < AbstractFile
  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
```

Таким образом, преимуществом данного подхода является отсутствие синтаксической избыточности, что позволяет использовать такие преимущества *DSL*, как самодокументированный код, простота понимания и расширяемость.

В тоже время недостатком данного подхода является потеря иерархии родительского автомата: состояние автомата не является отдельным классом, как было при использовании объектно-ориентированного подхода.

6. Протоколирование

Благодаря такому свойству динамических языков программирования, как метапрограммирование, 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
```

Данный класс отличается от того, который был использован в разделе 7 лишь одной строкой **`tracer`**. Это макровызов, который для каждого перехода и состояния выводит отладочную информацию, например:

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

7. Заключение

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

Достоинством использования объектно-ориентированного подхода к реализации наследования и вложения автоматов [2] является сохранение иерархии автомата при переносе его в объектно-ориентированный код благодаря тому, что каждое состояние и автомат являются отдельным классом.

Однако данный подход обладает таким недостатком, как синтаксическая избыточность и не позволяет легко модифицировать логику переходов.

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 этого подхода является потеря иерархии родительского автомата, так как состояние и автомат не являются отдельными классами, как было реализовано в объектно-ориентированном подходе.

Таким образом, оптимальным решением будет являться объединение объектно-ориентированного и динамического подходов: предметно-ориентированный язык будет использован для создания классов объектно-ориентированного подхода.

8. Список литературы

[1] Поликарпова Н. И., Шалыто А. А. Автоматное программирование. Учебно-методическое пособие. СПбГУ ИТМО. 2007. http://is.ifmo.ru/books/_umk.pdf

[2] Шопырин Д. Г., Шалыто А. А. Графическая нотация наследования автоматных классов // Программирование. 2007. № 5, с. 62–74. http://is.ifmo.ru/works/_12_12_2007_shopyrin.pdf

[3] Степанов О. Г., Шалыто А. А. Предметно-ориентированный язык автоматного программирования на базе динамического языка Ruby // Информационно-управляющие системы. 2007. № 4, с. 22–27. http://is.ifmo.ru/works/_2007_10_05_aut_lang.pdf

[4] 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. http://is.ifmo.ru/works/_astafurov_secr_word_2003.pdf

[5] Scott B. Acts As State Machine. http://agilewebdevelopment.com/plugins/acts_as_state_machine

[6] Thomas D., Fowler C., Hunt A. Programming Ruby. Second Edition. Texas: Pragmatic Bookshelf. 2004

[7] Parr T. The Definitive Antlr Reference: Building Domain-Specific Languages. Texas: Pragmatic Bookshelf. 2007

[8] Заякин Е. А., Шалыто А. А. Метод устранения повторных фрагментов кода при реализации конечных автоматов. СПбГУ ИТМО. 2007. http://is.ifmo.ru/projects/life_app/