УДК 004.424.2 DOI: 10.17587/prin.10.3-13

С. Б. Митькин, консультант, e-mail: stipan.mitkin@gmail.com, ProsessPilotene AS, Норвегия

Автоматное программирование на языке ДРАКОН

Предложен новый метод визуального автоматного программирования на языке ДРАКОН, состоящий в совмещении конечных автоматов с деревьями принятия решений. Описана генерация программного кода из автоматных ДРАКОН-схем и способ передачи сигналов между автоматами. Рассмотрен опыт применения предлагаемого метода в реальном программном проекте.

Ключевые слова: *автоматное программирование, конечные автоматы, машины состояний, язык ДРАКОН, реактивные системы, визуальное программирование, деревья принятия решений*

Введение

Автоматное программирование облегчает моделирование сложного поведения. В работе [1] авторы рекомендуют использовать автоматный подход при создании любой программной системы, в которой есть сущности со сложным поведением. Поведение системы, как правило, прямо связано с управлением этой системой. В работе [2] авторы отмечают, что задачи управления присутствуют в любом программном обеспечении, как следствие, конечные автоматы могли бы использоваться во всех современных программах.

К настоящему времени разработаны методы построения программ с применением конечных автоматов, например, SWITCH-технология [3]. Тем не менее, в программном обеспечении автоматы встречаются редко. Причина недооценки автоматного подхода — высокая трудоемкость автоматного программирования.

Для работы с автоматами существуют библиотеки для популярных языков программирования, например, [4] и [5]. Библиотеки снижают трудозатраты, однако создаваемые с помощью библиотек автоматы сложны для чтения и анализа ввиду отсутствия визуализации.

Графические нотации, например, диаграммы состояний UML (*UML state machine*), показывают структуру автоматов в более понятном виде. Существуют инструментальные средства, например [6—8], которые генерируют программный код из диаграмм состояний. Графика облегчает анализ автоматов и снижает трудоемкость разработки по сравнению с чисто текстовым автоматным программированием. К сожалению, большинство таких средств, включая [6] и [7], требуют от разработчика писать часть автоматного кода отдельно от диаграммы, что по-прежнему создает трудности.

Диаграммы состояний UML имеют ряд недостатков. Во-первых, элементы на таких диаграммах не упорядочены, что заставляет читателя прилагать дополнительные усилия, чтобы понять структуру автомата. Во-вторых, на диаграмме состояний трудно выявить, какие именно входные сигналы приводят к тем или иным действиям автомата. Кроме того, неупорядоченность в программах со сложным поведением порождает ошибки. В работе [2] F. Wagner отмечает: "Поведение — самая сложная часть программы и поэтому источник самых запутанных ошибок".

Для предотвращения ошибок и повышения производительности труда при разработке автоматов имеет смысл обратиться к визуально упорядоченному языку ДРАКОН. В литературе по языку ДРАКОН [9] вопрос о преодолении сложностей автоматного программирования не ставится и не рассматривается. В настоящей работе впервые предложен метод визуального автоматного программирования, основанный на языке ДРАКОН. Этот метод был реализован в среде разработки DRAKON Editor [10].

В настоящей статье не рассматриваются математические и алгоритмические свойства конечных автоматов. Цель статьи — показать новый способ графического изображения автоматов.

Введение в автоматное программирование и визуальный язык ДРАКОН

В данном разделе приведены общие сведения о языке ДРАКОН и об автоматном программировании. Конечные автоматы — математическая абстракция, но здесь и далее автоматы рассмотрены с точки зрения практической разработки программ. Основы языка ДРАКОН в данном разделе изложены без связи с автоматным программированием.

Введение в автоматное программирование

Широко распространено ошибочное мнение, что автоматное программирование — узкоспециальная область, ограниченная разработкой электромеханических автоматов или других аппаратных средств. В действительности автоматное программирование представляет собой подход к разработке любого программного обеспечения вне зависимости от его назначения. Этот подход основан на применении особого рода объектов — автоматов — в качестве строительных блоков. Так же как объектно-ориентированная программа построена из классов, автоматная программа состоит из автоматов.

Классы и автоматы имеют общие черты, а именно: те и другие хранят внутри себя данные; те и другие принимают сигналы извне и реагируют на них.

Несмотря на сходство, автоматы имеют отличия от классов.

- Среди данных, хранимых в автомате, выделяют особое поле, называемое *управляющим состоянием*. Значение этого поля принадлежит некоторому конечному множеству.
- Реакция автомата на входящий сигнал задается не только типом сигнала, но и значением управляющего состояния.

У обычного класса каждый метод представляет собой процедуру. У автомата каждому методу соответствует несколько процедур. Какая именно процедура будет выполнена при вызове данного конкретного метода, зависит от текущего управляющего состояния. Иными словами, выбор процедуры определяется комбинацией типа сигнала и управляющего состояния.

Далее в статье под термином "состояние" в единственном числе понимается текущее значение, которое хранится в управляющем состоянии автомата. Термин "состояния" во множественном числе означает перечень допустимых значений, которые может принимать управляющее состояние автомата.

Возьмем элемент графического интерфейса, меняющий цвет с белого на голубой при наведении на него мыши. Можно построить автомат, который управляет цветом этого элемента. Такой автомат будет иметь два возможных состояния: "активное" и "обычное". Автомат будет принимать два вида сигналов: "движение мыши над элементом" и "мышь покинула элемент". Начальное состояние — "обычное", начальный цвет элемента — белый.

Если автомат получит сигнал "движение мыши над элементом", находясь в состоянии "обычное", он поменяет цвет элемента на голубой и переключится в состояние "активное". Если этот сигнал придет, когда автомат уже находится в состоянии "активное", автомат проигнорирует сигнал.

Если автомат получит сигнал "мышь покинула элемент", находясь в состоянии "активное", автомат поменяет цвет элемента на белый и перейдет в состояние "обычное". Если сигнал "мышь покинула элемент" поступит, когда автомат находится в состоянии "обычное", это может свидетельствовать об ошибке в других подсистемах программы. Процедура, запускаемая в состоянии "обычный", по сигналу "мышь покинула элемент" может либо проигнорировать сигнал, либо сообщить об ошибке.

В этом примере автомат, управляющий цветом, обрабатывает сигналы по-разному в зависимости от своего состояния, которое зависит от предыдущих сигналов. Можно сказать, что автомат — это класс, который переключается между несколькими возможными режимами работы.

Текущее состояние автомата — результат цепочки предыдущих событий, произошедших с автоматом, поэтому состояние представляет собой связь с прошлым автомата. Эта связь указывается в явном виде, причем прошлое получается сжатым в одну переменную. Именно такая компактность представления прошлого делает автоматы удобным инструментом моделирования поведения.

Автоматный подход хорошо масштабируется, когда сложность поведения систем возрастает. Масштабируемость достигается построением сетей, состоящих из многих простых автоматов.

Автомат — мощная и в то же время простая программная модель. Автоматы могут помочь в решении задач реализации поведения в самых разных видах программного обеспечения.

Основы языка ДРАКОН

Язык ДРАКОН — визуальный язык представления алгоритмов. ДРАКОН-схемы имеют сходство с блок-схемами, однако в языке ДРАКОН есть ряд упорядочивающих правил, направленных на облегчение понимания диаграмм. Вот некоторые из этих правил.

- Пересечения линий запрещены.
- Разрешены только прямые вертикальные и горизонтальные линии.
- Течение времени на диаграмме направлено сверху вниз.
- Ветвление происходит только вправо.
- Разрешен только один вход в цикл.

В языке ДРАКОН есть визуальные аналоги конструкций структурного программирования — `if-else`, `switch-case`, `foreach-break` и т. п. Можно сказать, что язык ДРАКОН имеет такие же преимущества перед традиционными блок-схемами, как структурное программирование перед программированием на основе оператора `goto`.

Из ДРАКОН-диаграмм можно генерировать исполняемый или программный код. Для этого в иконах помещают выражения на некотором языке программирования. В настоящей работе в качестве такого языка программирования выбран язык JavaScript. В выражениях внутри икон не должно быть ключевых слов `if-else`, `switch-case` и т. п. Роль таких ключевых слов играют конструкции языка ДРАКОН.

Для линейных участков алгоритма применяют иконы "действие" (рис. 1).

Для вопросов, на которые можно ответить "да" или "нет", применяют икону "вопрос" (аналог оператора `if-else`, рис. 2).

Для вопросов, на которые может быть несколько ответов, применяют макроикону "выбор" (аналог оператора `switch-case`), состоящую из одной иконы "выбор" и двух или более икон "вариант". Макроикона "выбор" на рис. 3 имеет три иконы "вариант". Пустая икона "вариант" справа означает "все остальные значения".

В языке ДРАКОН есть тип диаграмм, называемый *силуэт*. Силуэт разбивает большой алгоритм на несколько малых алгоритмов. Каждый из таких малых алгоритмов заключен в отдельной *ветке силуэта*. Вверху ветки силуэта находится икона "имя ветки".

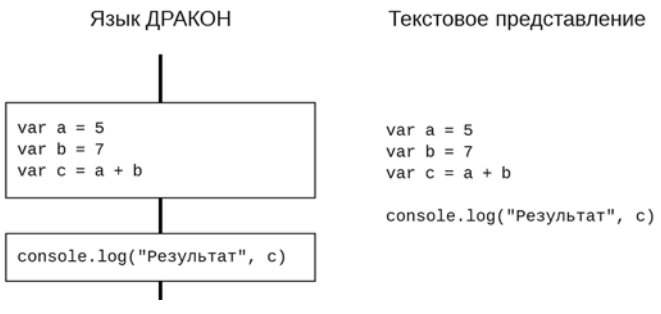


Рис. 1. Иконы "действие" на линейном участке алгоритма

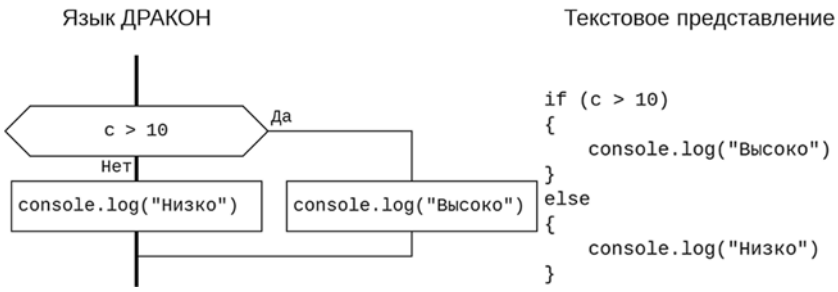
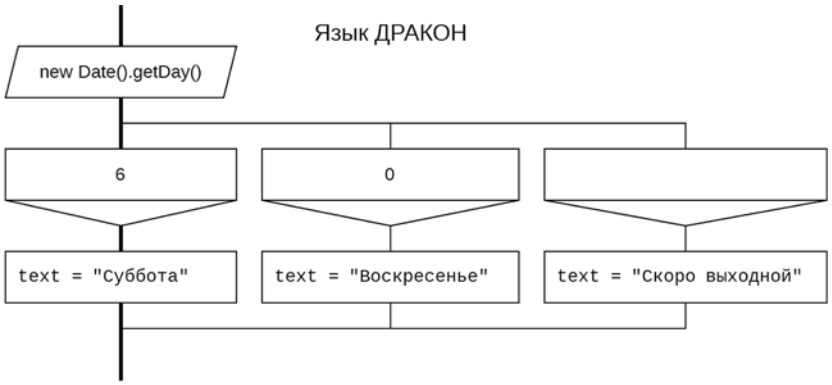


Рис. 2. Икона "вопрос" и ветвление алгоритма



Текстовое представление

```
switch (new Date().getDay()) {
    case 6:
        text = "Суббота"
        break
    case 0:
        text = "Воскресенье"
        break
    default:
        text = "Скоро выходной"
}
```

Рис. 3. Макроикона "выбор" и вопрос с несколькими ответами

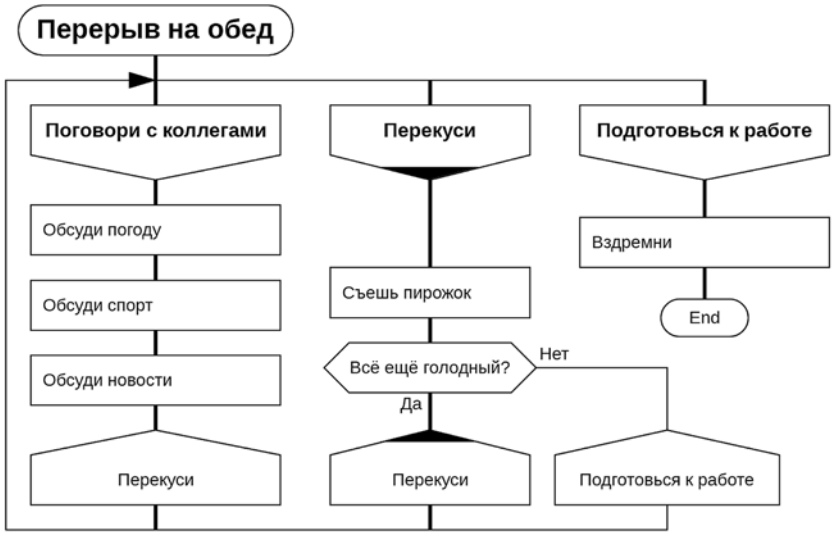


Рис. 4. Пример ДРАКОН-схемы "силуэт"

Ниже следует тело ветки, содержащее малый алгоритм — часть общего алгоритма. Внизу расположена одна или несколько икон "адрес", которые указывают на следующие ветки силуэта.

Самая первая ветка расположена слева. Порядок следования веток задается иконами "адрес", при этом существует соглашение упорядочивать ветки слева направо. Некоторые ветки могут выполняться несколько раз. Ветка с выходом из алгоритма, если выход есть, всегда находится справа. На рис. 4 представлена ДРАКОН-схема "силуэт", в которой есть три ветки: "Поговори с коллегами", "Перекуси", "Подготовься к работе".

Язык ДРАКОН представляет собой графический аналог структурного программирования, упорядоченный в целях повышения читаемости диаграмм. ДРАКОН-схемы пригодны для разработки программного обеспечения — из них можно получать работающие программы. Одной из особенностей языка ДРАКОН является тип диаграмм "силуэт", который разбивает задачу на логические части.

Изображение автоматов с помощью языка ДРАКОН

В данном разделе описана суть предлагаемого графического способа изображения автоматов. Вначале приведено формальное описание представления автоматов на языке ДРАКОН, затем дан пример автоматной ДРАКОН-схемы и программный код, сгенерированный из этой схемы. Для генерации кода применялась среда разработки DRAKON Editor, в которую автор добавил поддержку автоматных диаграмм.

Формальное описание изображения автоматов на языке ДРАКОН

В предлагаемом методе изображения автоматов применяется подмножество языка ДРАКОН:

- используются только схемы типа "силуэт", схемы "примитив" не используются;
- в теле ветки схемы "силуэт" допускаются следующие иконы языка ДРАКОН: "действие", "вопрос" и макроикона "выбор".

Графический синтаксис языка ДРАКОН остается без изменений. Способ расположения элементов на схеме не меняется. Меняется семантика схемы "силуэт".

Ветки на обычных, не автоматных схемах "силуэт" обозначают последовательно исполняемые участки алгоритма. Начало ветки (икона "имя ветки") не подразумевает остановки алгоритма.

Ветки на автоматных схемах "силуэт" означают возможные состояния автомата. Каждая ветка соответствует одному возможному состоянию автомата. Переход в начало ветки означает переключение в соответствующее состояние, приостановку алгоритма работы автомата и ожидание входящего сигнала в этом состоянии.

Пусть автомат M задается пятеркой следующих элементов:

$$M = \{V, Q, q_0, F, \delta\},$$

где V — входной алфавит; Q — множество возможных состояний; q_0 — начальное состояние; F — множество конечных состояний ($F \subset Q$); δ — функция переходов, δ: $V \times Q \to Q$.

Каждому состоянию $q_i \subset Q$ соответствует одна ветка силуэта. Самая левая ветка — начальное состояние q_0. Если силуэт не имеет ветки с иконой "конец", множество конечных состояний пусто. Если ветка с иконой "конец" есть, то такая ветка расположена справа на схеме. Данная конечная ветка означает единственное конечное состояние $f \subset Q$.

В начале каждой ветки, кроме ветки с иконой "конец", находится макроикона "выбор" с ключевым словом "receive". Эта макроикона ставится непосредственно под иконой "имя ветки". В ветке может быть только одна макроикона "выбор" с ключевым словом "receive".

Иконы "вариант" сразу под иконой "выбор" с ключевым словом "receive" задают типы входящих сигналов $v \subset V$, которые автомат принимает в данном состоянии q_i. Такие иконы "вариант", расположенные на разных ветках, могут задавать различные подмножества принимаемых сигналов. Непустые иконы "вариант" на всех ветках вместе задают входной алфавит, или множество входных символов (сигналов) V автомата. Пустая икона "вариант" означает "любой из сигналов, упомянутый на других ветках".

Тела веток силуэта задают функцию переходов δ: $V \times Q \to Q$, $\delta(v_j, q_i) \to q_k$, где входной символ v_j обозначается иконой "выбор", входное состояние q_i обозначается иконой "имя ветки", а выходное состояние q_k — иконой "адрес".

Участок диаграммы под каждой иконой "вариант" представляет собой обработчик одного типа сигнала в состоянии, соответствующем данной ветке. Обработчики сигналов задают действия автомата при получении сигналов. В обработчиках сигналов могут быть иконы "действие", а также средства ветвления — иконы "вопрос" и макроиконы "выбор" без ключевого слова "receive". Благодаря ветвлению, из одной иконы "вариант" можно попасть в несколько икон "адрес", поэтому обработчики сигналов представляют собой деревья принятия решений.

Таким образом, при данном способе визуализации наиболее важные компоненты автомата присутствуют на одном чертеже: перечень состояний автомата задается именами веток силуэта, входной алфавит определяется иконами "выбор", а функция переходов содержится в телах веток.

Пример автоматной ДРАКОН-схемы и сгенерированного из нее кода

Среда разработки DRAKON Editor генерирует исходный код из ДРАКОН-диаграмм. По состоянию на октябрь 2018 г. поддерживается 13 языков программирования.

В обычном режиме из одной ДРАКОН-схемы получается одна функция. Для некоторых языков, включая JavaScript, Python, C# и Lua, DRAKON Editor генерирует не только обычные функции, но и конечные автоматы. Сгенерированные автоматы построены в виде классов на выбранном языке программирования. Из одной автоматной ДРАКОН-схемы получается один класс — один автомат.

На рис. 5 представлена автоматная ДРАКОН-схема на гибридном языке ДРАКОН-JavaScript. Чтобы указать, что ДРАКОН-схема является автоматной, в первой строке иконы "параметры" помещают ключевую фразу "state machine" (см. прямоугольник в левой верхней части рис. 5). DRAKON Editor строит автоматы только из ДРАКОН-схем "силуэт". Для схем "примитив" генерация автоматного кода не предусмотрена.

Имя класса берется из названия автоматной ДРАКОН-схемы. Названия состояний берутся из названий веток. В названии схемы и названиях веток не должно быть пробелов или символов пунктуации, так как эти названия войдут в состав идентификаторов в сгенерированной программе.

В состоянии `Foo` автомат обрабатывает сигналы `yellow` и `red`, а в состоянии `Bar` — `yellow` и `black`. Таким образом, автомат `Abc` принимает три вида сигналов: `yellow`, `red` и `black`. В состоянии `Exit` автомат выключен и не принимает сигналов.

DRAKON Editor сгенерирует из этой диаграммы класс `Abc` с тремя методами: `yellow`, `red` и `black`. Все эти методы имеют одинаковую сигнатуру — они принимают один аргумент `msg`, который задан в иконе "Параметры". Клиентский код может использовать класс `Abc` так:

```
var machine = new Abc()
machine.yellow(msg1)
machine.black(msg2)
machine.red(msg3)
```

Рассмотрим состояние `Foo` (первое слева на рис. 5). В состоянии `Foo` автомат принимает сигналы `yellow` и `red`. Если вызвать метод `black` в этом состоянии, ничего не случится. Вызов будет проигнорирован без генерации исключения. Важно иметь возможность игнорировать ненужные сигналы. Если в автомат приходит сигнал, который не обрабатывается в текущем состоянии, это не ошибка. Игнорирование сигналов — полезный элемент поведения автоматов. Если же приход некоторого сигнала считается ошибкой, следует создать для этого сигнала обработчик и предпринять там соответствующие явные действия.

Обработчик сигнала `yellow` в состоянии `Foo` вызывает функцию `goLeft(self, msg)` и переключает автомат в состояние `Bar`, если значение `msg.x` больше 100. Если значение `msg.x` меньше или равно числу

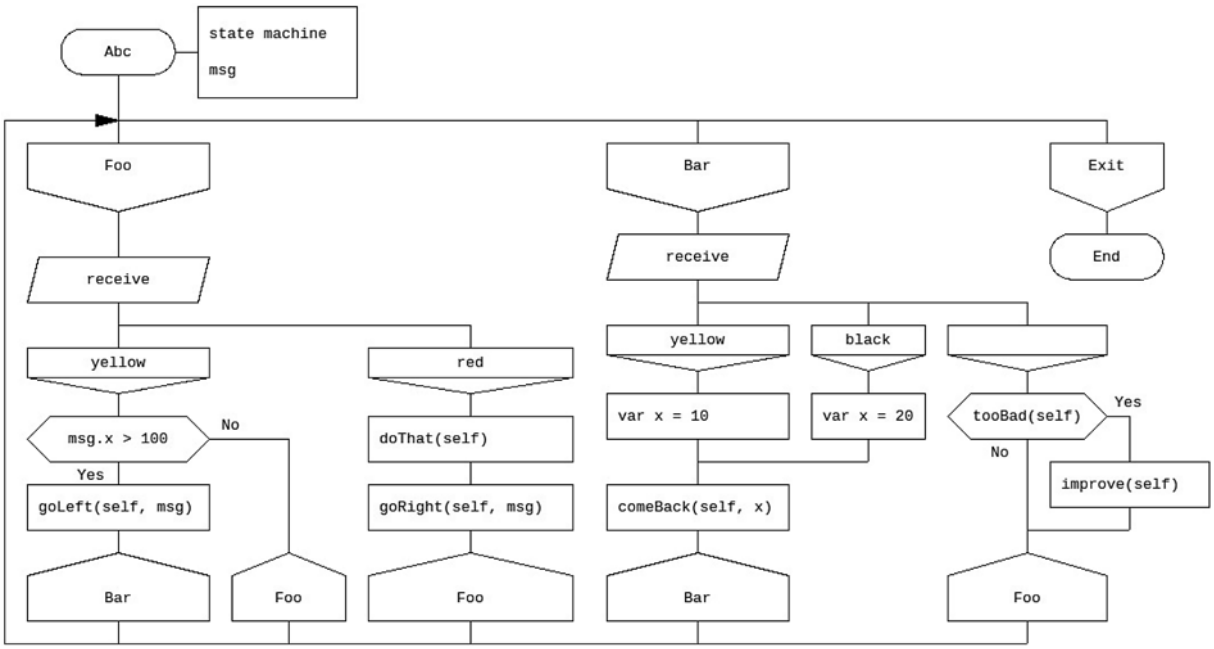


Рис. 5. Схема конечного автомата `Abc`, который имеет состояния `Foo`, `Bar` и `Exit`

100, обработчик сигнала `yellow` не совершает никаких действий. Обработчик сигнала `red` вызывает функции `doThat(self)` и `goRight(self, msg)` и не переключает состояние автомата. `goLeft`, `doThat`, `goRight` и другие функции должны быть определены в другом месте, вне автомата `Abc`.

Переменная `self` указывает на объект автомата, она доступна во всех обработчиках.

Перейдем к состоянию `Bar` (средняя ветка на рис. 5). В состоянии `Bar` автомат обрабатывает сигналы `yellow` и `black`. Хотя пути обработчиков сигналов `yellow` и `black` соединяются внизу диаграммы, DRAKON Editor построит разные функции для этих сигналов, причем совпадающий участок алгоритмов будет скопирован. Поэтому локальная переменная `x` должна быть объявлена два раза — по одному разу в каждом обработчике.

Пустая икона "вариант" означает "все остальные сигналы". Для состояния `Bar` "все остальные" — это сигнал `red`. При вызове метода `red` в состоянии `Bar` будет запущен обработчик, который начинается с пустой иконы "вариант".

Пустую икону "вариант" можно также использовать, когда в некотором состоянии автомат принимает только один вид сигналов. Макроикона "выбор" с одной иконой "вариант" запрещена, но если к единственной иконе "вариант" добавить пустую икону "вариант", генератор кода посчитает такую макроикону "выбор" корректной.

Генератор кода анализирует ДРАКОН-схему и составляет список состояний и список типов принимаемых сигналов (входной алфавит автомата).

Список состояний для автомата `Abc` на рис. 5: `Foo`, `Bar`, `Exit`.

Список типов сигналов для автомата `Abc`: `black`, `red`, `yellow`.

Затем генератор создает тело класса, в который для каждого типа сигналов добавляет метод. В теле метода происходит выбор конкретного обработчика исходя из текущего состояния. В итоге обработчик определяется исходя из двух входных данных: типа сигнала и текущего состояния (*double dispatch*). Все обработчики принимают одни и те же аргументы. Это аргумент `self` и аргументы, перечисленные в иконе "параметры" ниже строки с ключевой фразой "state machine".

Генератор добавляет в класс поле `state`, которое хранит название текущего состояния автомата в виде строки. При создании автомата в поле `state` записывается начальное состояние автомата.

Строковое поле `state` удобно для тестирования и отладки. Если поле `state` содержит `null`, автомат считается выключенным и не реагирует на сигналы. Автомат можно выключить в клиентском коде принудительно, если записать `null` в поле `state`:

```
machine.state = null
```

Возможны и другие, более эффективные, способы хранения текущего состояния и выбора конкретного обработчика. Данный способ — один из наиболее простых в языках JavaScript и Lua.

Если на ДРАКОН-схеме в каком-то состоянии есть пустая икона "вариант", то для тех типов сигналов, которые не указаны явно в иконах "вариант" для данного состояния, будет сгенерирован и вызван обработчик по умолчанию. Например, в состоянии `Bar` класса `Abc` явно указаны события `yellow` и `black`, а событие `red` не указано, но есть пустая икона "вариант". Поэтому в методе `red` для состояния `Bar` вызывается обработчик `Bar` по умолчанию: `Abc _ Bar _ default`.

Тело класса для автомата Abc:

```
function Abc() {
   var _self = this;
   _self.type_name = "Abc";
   _self.state = "Foo";
   _self.black = function(msg) {
      var _state_ = _self.state
      if (_state_ == "Bar") {
         Abc_Bar_black(_self, msg)
      }
   }
   _self.red = function(msg) {
      var _state_ = _self.state
      if (_state_ == "Foo") {
         Abc_Foo_red(_self, msg)
   }
      else if (_state_ =="Bar") {
         Abc_Bar_default(_self, msg)
      }
   }
   _self.yellow = function(msg) {
      var _state_ = _self.state
      if (_state_ == "Foo") {
         Abc_Foo_yellow(_self, msg)
   }
      else if (_state_ == "Bar") {
         Abc_Bar_yellow(_self, msg)
      }
   }
}
```

Для каждого из сочетаний состояние — сигналы, которые присутствуют на диаграмме, генератор создает функцию-обработчик. В конце обработчика генератор добавляет код, который переключает автомат в следующее состояние. Начиная с версии генератора 1.32, обработчики могут возвращать значение с помощью ключевого слова return.

Пример: обработчик сигнала yellow для состояния Foo в классе Abc:

```
function Abc_Foo_yellow(self, msg) {
   if (msg.x > 100) {
      goLeft(self, msg)
      self.state = "Bar"
   } else {
      self.state = "Foo"
   }
}
```

Как видно из примера, из ДРАКОН-схемы автоматически генерируется не только базовая структура автомата, но и код обработчиков сигналов, включая алгоритм выбора следующего состояния. Генератор кода формирует весь исходный код автомата, так что нет необходимости дописывать код вручную.

Особенности применения автоматного программирования

Автор использовал средства визуального автоматного программирования среды DRAKON Editor при создании коммерческого веб-приложения DrakonHub [11]. К сожалению, создать приложение полностью в автоматном стиле не получилось, так как эти средства появились, когда часть кода данного веб-приложения уже была написана. Тем не менее, автору удалось опробовать предлагаемый метод на практике и получить обратную связь от применения этого метода в реальном программном проекте. Данный раздел содержит описание опыта, полученного в ходе выполнения этого проекта.

Решаемые задачи

Исходный код веб-приложения DrakonHub насчитывает 51 конечный автомат на языке ДРАКОН. Автоматное программирование на языке ДРАКОН показало свою пригодность для решения перечисленных далее задач.

- Асинхронный ввод/вывод, например, вызов веб-сервисов.
- Обработка действий пользователя, например, событий мыши, сенсорного экрана и клавиатуры.
- Мастера (wizards), т. е. графические диалоги, состоящие из последовательности экранов. Пример: диалог создания диаграммы, который состоит из двух шагов: выбора типа диаграммы и ввода названия диаграммы.
- Работа с таймером и управление длительными процессами. Пример: поиск по содержимому во многих диаграммах.
- Управление событиями в графическом интерфейсе пользователя, включая события нескольких связанных виджетов со сложным поведением. Пример: координация виджета редактора диаграмм и событий на сервере при одновременном редактировании диаграмм несколькими пользователями.
- Обход деревьев и графов. Пример: обход элементов диаграммы для поиска возможных мест, куда можно пересадить лиану (ребро ДРАКОН-схемы) без пересечения линий.
- Лексический анализ (разбор строк на лексемы).

Код приложения DrakonHub является закрытым. Для верификации практической применимости предлагаемого подхода можно обратиться к демонстрационному приложению Лифт [12].

Автоматное программирование на языке ДРАКОН хорошо себя зарекомендовало в событийно-ориентированной среде браузера и сервера приложений tarantool. Автор предполагает, что автоматы на основе ДРАКОН-схем будут также успешно работать и в других окружениях, например, в сервере приложений NodeJs.

Сеть автоматов

Одиночный автомат полезен только для ограниченного круга задач, например, для некоторых случаев обхода графов и простого лексического анализа. Чаще всего задачу решают с помощью нескольких взаимодействующих автоматов. Разбиение программы на множество автоматов позволяет работать с простыми и понятными автоматами, каждый из которых легко разработать и протестировать.

Существуют графические нотации, изображающие несколько автоматов на одной диаграмме, например, диаграммы состояний UML с иерархическими автоматами [13]. Проблемный вопрос с такими нотациями состоит в том, что сложность моделируемой системы ограничена размерами диаграммы.

Предлагаемый способ автоматного программирования не изображает дочерние автоматы на одной диаграмме с родительским. Граф автоматов строится отдельно, вне автоматных диаграмм, и как следствие, может быть довольно большим. Благодаря этому на размер моделируемой системы не налагается искусственных ограничений.

Способ организации взаимодействия автоматов имеет существенное значение. В процессе разработки приложения DrakonHub стало ясно, что сеть автоматов без ограничений на топологию трудно отлаживать. Практически действенным оказалось решение, описанное в работе [2]: упорядочить автоматы в виде дерева. Объединять все автоматы в программе в одно дерево не обязательно. Можно иметь набор деревьев или лес автоматов.

Требование выстраивать автоматы в виде дерева в некоторых случаях представляется слишком жестким. Сама необходимость упорядочивать автоматы возникает вследствие того, что во время выполнения некоторого метода автомата `A` другой автомат `B` может вызвать этот же или другой метод автомата `A`. Это может привести к ошибкам в силу разрушения инкапсуляции. Такая нежелательная ситуация возникает как с автоматами, так и с классами. Чтобы избежать этой ситуации, достаточно того, что автоматы образуют ориентированный граф без циклов. Те автоматы, которые находятся "выше" в графе, могут напрямую вызывать методы своих непосредственных соседей "ниже по течению" графа. Автоматы, которые находятся "ниже" в графе, посылают сигналы своим "верхним" соседям только опосредованно, через цикл событий (*event loop*).

Повышение сложности системы должно выражаться не в увеличении размера автоматов, а в росте их числа. Автоматы объединяются в сеть, причем важно, что сеть автоматов упорядочена в виде ориентированного графа без циклов. Каждый автомат изображается на отдельной диаграмме, а автоматная сеть строится вне автоматных диаграмм. Такой подход хорошо масштабируется и позволяет создавать системы с высоким уровнем сложности.

Размер диаграмм

Автоматные диаграммы в реальных приложениях могут быть большими, более 4 тыс. пикселей по горизонтали. Чтобы увидеть весь автомат, приходится прокручивать ДРАКОН-схему. Высота же автоматных диаграмм "силуэт" остается небольшой, не больше высоты экрана. Кроме того, в большинстве случаев можно добиться того, что одно состояние автомата полностью помещается на экране. Таким образом, значительный размер диаграмм не является препятствием.

Привлекательность автоматного программирования состоит в том числе и в том, что объект со сложным поведением разбивается на небольшие, относительно несложные части. Причем в один момент времени можно рассматривать только одну часть. Автоматные ДРАКОН-схемы способствуют именно такому подходу, так как элементы, связанные с одним состоянием, расположены в одной области диаграммы.

Анализ автоматов, изображаемых на языке ДРАКОН

Существуют различные виды автоматов и различные способы организации их работы. В данном разделе подчеркнуто отличие предлагаемого метода от других способов визуализации конечных автоматов, указано место ДРАКОН-автоматов в классификации автоматов, приведен перечень автоматных событий, или действий, которые можно отобразить на ДРАКОН-схеме, а также рассмотрена работа ДРАКОН-автоматов в средах, основанных на событиях, и в средах, проводящих регулярный опрос своих компонентов.

Деревья принятия решения

Отличительная особенность автоматных диаграмм на языке ДРАКОН — сочетание деревьев принятия решений и автоматной логики.

На автоматной ДРАКОН-схеме выбор реакции автомата делается в два шага. Первый шаг — тот же, что и в диаграммах состояний UML. Исходя из текущего состояния и входного сигнала выбирается обработчик сигнала. Второй, дополнительный шаг выбора осуществляется деревом принятия решений, которое находится в обработчике. Например, при поступлении в автомат на рис. 5 сигнала `yellow` в состоянии `Foo` сначала выбирается обработчик в левой нижней части диаграммы. Затем обработчик в зависимости от значения `msg.x` либо вызывает процедуру `goLeft` и переключает автомат в состояние `Bar`, либо оставляет автомат в состоянии `Foo`.

В литературе автоматы часто упоминаются вместе с деревьями принятия решений: деревья противопоставляются автоматам; деревья служат для иллюстрации автоматов; автоматы применяются для обхода деревьев и т. п. Новизна данного метода состоит в совместном использовании автоматов и деревьев принятия решений на одной диаграмме.

Автоматы Мили

Конечные автоматы, изображаемые на языке ДРАКОН, можно отнести к автоматам Мили. Выходная последовательность автомата Мили зависит от состояния автомата и входных сигналов. Именно так и работают автоматы, построенные в среде DRAKON Editor. Выбор обработчика происходит на основании двух факторов: типа входящего сигнала и состояния автомата.

Действия автомата, которые можно изобразить на ДРАКОН-схеме

Некоторые авторы [2] выделяют следующие автоматные события — действия, которые автомат выполняет за один цикл работы.

- **Входное действие** — процедура, которая выполняется, когда автомат входит в состояние. Для каждого состояния может быть одно входное действие.
- **Выходное действие**, которое выполняется, когда автомат покидает состояние. Для каждого состояния может быть одно выходное действие.
- **Действия перехода**, которые уникальны для каждой направленной пары состояний.
- **Действие обработки ввода**, которое автомат проводит при получении сигнала. Действия обработки ввода разные для разных комбинаций состояния и типов входного сигнала.

Программа DRAKON Editor версии 1.31 не поддерживает входные действия на автоматных ДРАКОН-схемах. Тем не менее принципиальная возможность добавить эти действия существует. Входные действия можно расположить над макроиконой "выбор" с ключевым словом "receive".

Выходные действия автоматные ДРАКОН-схемы не поддерживают совсем.

Действия перехода поддерживаются, но лишь частично. Часть алгоритма прямо над иконой "адрес" будет выполнена перед переключением из текущего состояния в следующее. Однако из одного состояния в другое может вести несколько икон "адрес". Это не соответствует требованию к действиям перехода быть уникальными для каждой пары состояний.

Автоматные диаграммы на языке ДРАКОН поддерживают действия обработки ввода в полной мере. Участки ДРАКОН-схемы, которые начинаются от икон "выбор" сразу под словом "receive", задают отдельные самостоятельные алгоритмы. Эти алгоритмы запускаются входящими сигналами. Комбинация состояния и типа сигнала определяет, который из таких алгоритмов следует выполнить.

На автоматной ДРАКОН-схеме можно изобразить только два из четырех автоматных действий. Это является ограничением предлагаемого метода. Вместе с тем самое важное для многих автоматов действие — действие обработки ввода — с помощью языка ДРАКОН отображается.

Событийно-ориентированное программирование и регулярный опрос

Автоматы могут работать в различных режимах, включая следующие.

- Событийно-ориентированное программирование ("реактивные" системы). В среде, которая реагирует на события, обработчик сигнала запускается, только когда приходит сигнал.
- Регулярный опрос (*polling*). Среда регулярно опрашивает автомат, вызывая процедуру обновления. Процедура обновления автомата проверяет наличие входных условий и, если условия выполняются, проводит соответствующие действия.

Автоматы в среде DRAKON Editor изначально создавались для событийно-ориентированного программирования, где при возникновении событий среда посылает автоматам сигналы. Чтобы послать сигнал автомату, в обработчике события нужно вызвать метод автоматного объекта для данного типа сигналов.

Работа в режиме регулярного опроса также возможна. Для этого автоматные ДРАКОН-программы должны принимать один сигнал `update`, обработчик которого и будет процедурой обновления автомата.

Таким образом, с помощью языка ДРАКОН можно создавать автоматы как для сред, основанных на событиях, так и для сред, работающих в режиме регулярного опроса. Однако наиболее удобно применять предлагаемый метод в рамках событийно-ориентированного программирования.

Преимущества автоматного программирования на языке ДРАКОН

Предлагаемый метод визуализации автоматов был создан в целях облегчения разработки программ в автоматном стиле. В данном разделе приведено детальное описание выгод, которые приносит представление автоматов на языке ДРАКОН.

Преимущества деревьев принятия решений

Деревья принятия решений позволяют получить следующие полезные эффекты.

- Увеличение выразительной силы диаграмм. ДРАКОН-схема позволяет чередовать проверку условий и действия в дереве принятия решений (рис. 6). Такое чередование нельзя изобразить на диаграмме состояний UML.
- Возможность визуально проследить путь через дерево принятия решений. Эта возможность увеличивает наглядность автоматной диаграммы, облегчает работу с ней и снижает риск возникновения ошибок.
- Помощь в устранении противоречивых условий. Условия перехода между состояниями на диаграмме состояний UML записывают в виде текстовых логических формул. При этом можно указать неполные или противоречащие условия. Особенно легко допустить такую ошибку, когда условия состоят из сложных логических выражений (рис. 7). Деревья принятия решений ввиду своей визуальной природы сводят возможность таких ошибок к минимуму.
- Привлекательность для широкого круга разработчиков. Многие разработчики не принимают автоматное программирование в силу того, что оно слишком сильно отличается от привычных методов создания программ. Деревья принятия решений близки к традиционному структурному программированию, и поэтому автоматное программирование на языке ДРАКОН имеет шансы на более широкое распространение, чем диаграммы состояний UML.

Эти преимущества усиливаются тем, что деревья принятия решений в автоматных ДРАКОН-схемах изображаются на одной визуальной сцене с другими частями автомата. Не требуется переключаться между разными документами и типами диаграмм, чтобы полностью охватить автомат.

Наличие деревьев принятия решений повышает выразительность автоматных диаграмм. Кроме того, деревья принятия решений делают автоматы более легкими для понимания.

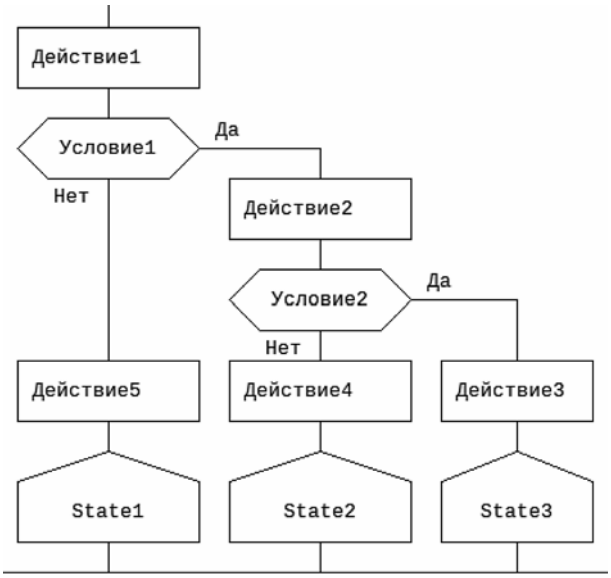


Рис. 6. Дерево принятия решения, в котором чередуются условия и действия

Избавление от "технических" состояний, не несущих смысла для читателя диаграммы

Возможность чередовать проверку условий и действия в деревьях принятия решений дает дополнительный выигрыш: алгоритм обработчика может сделать несколько последовательных вызовов методов дочерних автоматов. В результате появляется возможность избавиться от многих "технических" состояний, которые образуются при ожидании ответа от дочерних автоматов.

Например, автомат `Scheduler` (планировщик) из работы [12] имеет два рабочих состояния: `Up` и `Down`. Эти состояния соответствуют двум направлениям движения кабины лифта: вверх и вниз. Нет необходимости добавлять дополнительные состояния для ожидания ответа от дочернего автомата `Buttons`.

Благодаря этому каждое состояние автомата является конкретным наблюдаемым режимом работы. Такие конкретные режимы работы легко выявлять на этапе разработки автоматов.

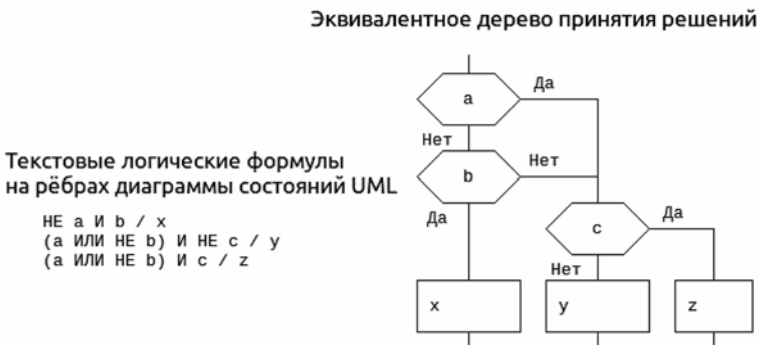


Рис. 7. Текстовые логические формулы на ребрах диаграммы состояний UML и эквивалентное им дерево принятия решений

Наглядность автоматов

Правила языка ДРАКОН эргономически выверены, и поэтому делают автоматные диаграммы легкими для прочтения. Например, чтобы определить, из каких веток состоит силуэт, не нужно изучать всю диаграмму. Для этого достаточно просмотреть самый верх схемы, так как заголовки веток выровнены по горизонтали и расположены вверху. В результате одного взгляда на верх автоматной ДРАКОН-схемы достаточно, чтобы увидеть состояния автомата.

Для сравнения: чтобы увидеть состояния на диаграмме состояний UML, схему требуется просмотреть полностью.

Иконы "вариант", задающие типы сигналов, всегда находятся вверху ветки, и это тоже облегчает чтение автоматной диаграммы. Во-первых, набор принимаемых сигналов оформлен в виде списка, который выстроен по горизонтали. Во-вторых, читатель заранее знает, где искать этот список — наверху ветки.

Для сравнения: чтобы увидеть входные сигналы на диаграмме состояний UML, необходимо отследить все ребра, исходящие из иконы выбранного состояния, причем ребра могут идти в произвольных направлениях. Затем нужно найти подпись на каждом ребре и выделить из нее условие.

Иконы "адрес" выровнены по горизонтали и расположены внизу диаграммы. Благодаря этому можно узнать, в какие состояния автомат может перейти из любого выбранного состояния, без изучения всей ДРАКОН-схемы.

Обработчики сигналов обязательно заканчиваются иконами "адрес". Данная нотация исключает возможность забыть указать следующее состояние в алгоритме обработчика.

Так как начальное состояние на автоматной ДРАКОН-схеме всегда расположено слева, алгоритм инициализации, если он есть, тоже расположен слева. Это — еще один фактор, который способствует единообразию автоматных диаграмм.

Язык ДРАКОН налагает на диаграммы строгие правила. Одно из таких правил — запрет на пересечение линий. Отсутствие пересечений устраняет целый класс запутанных диаграмм и связанных с ними ошибок.

Другое правило состоит в том, что время в ветке силуэта течет сверху вниз, а ветвление идет только вправо. Это правило дает деревьям принятия решений предсказуемость. Читатель находит следующий шаг алгоритма там, где ожидает.

Кроме того, упорядоченность ветки силуэта в направлении сверху вниз вводит стандартную последовательность прочтения состояния в автомате.

- На самом верху — название состояния.
- Чуть ниже — сигналы, которые автомат принимает в этом состоянии.
- Еще ниже — обработчики сигналов.
- В самом низу — следующие состояния, в которые можно перейти из данного состояния.

Несмотря на то, что автоматные ДРАКОН-схемы могут достигать значительных размеров по ширине, каждое конкретное состояние обычно помещается на экран. Все элементы автомата, связанные с одним состоянием, расположены недалеко друг от друга на диаграмме. Это облегчает концентрацию внимания на одном состоянии в один момент времени, что позволяет применять автоматное программирование максимально эффективно.

Наглядность — понятие во многом субъективное, однако описанные выше меры повышения читаемости автоматных диаграмм действуют вне зависимости от индивидуальных предпочтений разработчиков. Упорядоченное расположение элементов на схеме воспринимается лучше, чем хаотичное, а графическая группировка частей автомата по принадлежности к состоянию помогает чтению диаграммы больше, чем случайный разброс этих частей. Все вместе эти упорядочивающие меры повышают комфорт разработчика и сокращают вероятность ошибок.

Отсутствие необходимости дописывать автоматный код вручную

Чем больше рабочего программного кода можно получить из диаграммы, тем лучше. И наоборот, чем больше кода приходится дописывать вручную, тем менее очевидна выгода от визуального программирования.

Многие инструменты визуального автоматного программирования, например [6] и [7], генерируют лишь часть кода, составляющего автомат. Эти инструменты производят только код, который составляет структуру автомата. Содержательную часть автомата, например, обработчики сигналов, программист должен написать в текстовом виде в отдельном файле.

Порядок работы с такими инструментами таков.

- Разработчик рисует автоматную диаграмму.
- Инструмент генерирует код структуры автомата.
- Разработчик пишет код реакций автомата в текстовых файлах. Связывание написанного кода со структурой автомата происходит либо неявно, на основе соглашений по именованию функций или классов, либо явно, с помощью связующего кода.

Для автоматных ДРАКОН-программ порядок работы иной.

- Разработчик рисует автоматную диаграмму.
- Инструмент генерирует весь код автомата. Нет необходимости писать и связывать дополнительный код — автомат готов к работе.

Таким образом, автоматное программирование на языке ДРАКОН дает очевидную выгоду: разработчик более не вынужден редактировать дополнительный файл с исходным кодом при создании и доработке автомата. Такая выгода повышает удобство и производительность труда разработчика.

Визуальная подсказка в тестировании

Одна из сильных сторон автоматного программирования — ясная стратегия тестирования. Большая программа со сложным поведением строится как система автоматов. Каждый автомат тестируется отдельно. В автомате последовательно перебирают состояния и проверяют реакции автомата в каждом из состояний.

ДРАКОН-схема предоставляет визуальную помощь в следовании этой стратегии. Чтобы перебрать состояния автомата, достаточно прочитать заголовки веток силуэта. Это просто, так как заголовки веток выровнены, как горизонтальный список. Чтобы перебрать типы входящих сигналов для выбранного состояний, следует пройтись по иконам "вариант" вверху соответствующей ветки. Иконы "вариант" — это тоже горизонтальный список. Одна икона "вариант" — один обработчик входящих сигналов.

Далее нужно составить тестовые сценарии для каждого пути через обработчик. Обработчик на ДРАКОН-схеме представляет собой дерево принятия решений. Чтобы построить тест, нужно визуально проследить каждый путь через дерево и записать шаги, из которых состоит этот путь. Графическое представление дерева делает такое прослеживание тривиальным.

Эти визуальные подсказки помогают при написании тестов и облегчают тестирование. Легкость тестирования способствует более полному покрытию программы тестами, что повышает надежность программы.

Заключение

Предложен новый метод визуального автоматного программирования на языке ДРАКОН. Данный способ повышает производительность труда разработчика и имеет ряд преимуществ по сравнению с другими методами построения автоматов.

ДРАКОН-схема "силуэт" изображает на одной визуальной сцене все части автомата: состояния, типы принимаемых сигналов, обработчики сигналов и переходы. При этом относящиеся к одному состоянию элементы находятся в одной области на диаграмме. Это облегчает анализ и редактирование автоматов, а также сокращает число ошибок.

Автоматная ДРАКОН-схема содержит всю необходимую информацию для генерации программного кода. Из ДРАКОН-схемы можно сгенерировать не только код, который задает структуру автомата, но и содержательный код обработчиков. Не требуется вручную дописывать исходный код автомата в отдельном файле.

Достоинство автоматных ДРАКОН-схем — предсказуемость и единообразие, которые улучшают читаемость диаграмм. Язык ДРАКОН предоставляет соглашение, которому должны следовать диаграммы. Это соглашение упорядочивает автоматы и дает автоматам единую структуру.

Принципиальное отличие автоматных ДРАКОН-схем от других графических нотаций для автоматного программирования состоит в совмещении автоматного выбора действий с деревьями принятия решений. Деревья принятия решений делают автоматы более гибкими и понятными для широкого круга разработчиков.

Благодаря деревьям принятия решений сокращается число "технических" состояний автомата, не несущих смысла для человека, что облегчает проектирование программ.

Автоматные ДРАКОН-программы просто тестировать, так как ДРАКОН-схема дает визуальную подсказку при составлении тестовых сценариев.

Предлагаемый метод автоматного программирования, основанный на языке ДРАКОН, снижает

трудоемкость создания автоматов и сокращает число ошибок. Благодаря языку ДРАКОН работа с автоматами облегчается настолько, что разработчик может применять автоматы везде, где к этому располагает задача. Автоматное программирование становится более доступным и эффективным.

Список литературы

1. **Поликарпова Н. И., Шалыто А. А.** Автоматное программирование. СПб.: Питер, 2011. 176 с.
2. **Wagner F., Schmuki R., Wagner T., Wolstenholme P.** Modeling Software with Finite State Machines. New York: Auerbach Publications, 2006. 392 p.
3. **Шалыто А. А.** SWITCH-технология. Алгоритмизация и программирование задач логического управления. СПб: Наука, 1998. URL: http://is.ifmo.ru/books/switch/1
4. **Machina.js.** URL: http://machina-js.org/
5. **Boost** Meta State Machine. URL: https://www.boost.org/doc/libs/1_67_0/libs/msm/doc/HTML/index.html
6. **UniMod.** URL: http://unimod.sourceforge.net/
7. **Rosmaro**. URL: https://rosmaro.js.org/
8. **YAKINDU** Statechart Tools. URL: https://www.itemis.com/en/yakindu/state-machine/
9. **Пароңджанов В. Д.** Учись писать, читать и понимать алгоритмы. М.: ДМК Пресс, 2012. 520 с.
10. **DRAKON** Editor. URL: http://drakon-editor.sourceforge.net/
11. **DrakonHub.** URL: https://drakonhub.com/
12. **Митькин С. Б.** An interactive state machine demo. URL: https://drakonhub.com/files/lift.html
13. **Miro Samek.** Introduction to Hierarchical State Machines, 2016, URL: https://barrgroup.com/Embedded-Systems/How-To/Introduction-Hierarchical-State-Machines

Automata-Based Programming in DRAKON Language

S. Mitkin, Consultant, stipan.mitkin@gmail.com, ProsessPilotene AS, Asker, 1383, Norway

Corresponding author:

Mitkin Stepan, Consultant, ProsessPilotene AS, Asker, 1383, Norway,
E-mail: stipan.mitkin@gmail.com

Received on September 09, 2018
Accepted on October 23, 2018

A new method for visualization of finite automata (state machines) is proposed. This method combines automata with decision trees using the DRAKON algorithmic language.

DRAKON has visual branching and looping constructs that correspond to the building blocks of structured programming: if/then/else, switch/case, while/for. DRAKON's branching constructs follow strict ergonomic guidelines that help one build consistent and readable decision trees. Another feature of DRAKON is "silhouette" diagrams. A silhouette splits a large algorithm into smaller parts which are called "branches."

In the proposed method, the branches of a silhouette designate the states of an automaton. Each branch has all the information on the corresponding state including its name, the state-specific input symbols, the names of the next states. The branch also contains decision trees that represent the input symbol processing algorithms and state-switching logic. This kind of state machine diagrams is suitable for programming—the author has built a software tool that generates working programs from automata diagrams based on DRAKON.

The presence of decision trees on state machine diagrams produces several positive effects, such as a higher expression power and better readability of automata. Besides, the DRAKON language brings a standardized layout that makes diagrams predictable and easy to comprehend. These benefits together make automata more comfortable to work with and reduce the number of errors. As a result, automata-based programming becomes more practical and productive.

Keywords: *automata-based programming, finite automaton, state machine, DRAKON language, event-driven system, visual programming, decision tree, software system behavior, state diagram*

For citation:

Mitkin S. Automata-Based Programming in DRAKON Language*, Programmnaya Ingeneria,* 2019, vol. 10, no. 1, pp. 3—13

DOI: 10.17587/prin.10.3-13

References

1. **Polikarpova N. I., Shalyto A. A.** *Avtomatnoe programmirovanie* (Automata-based programming), Saint-Petersburg, Piter, 2011, 176 p. (in Russian).
2. **Wagner F., Schmuki R., Wagner T., Wolstenholme P.** Modeling Software with Finite State Machines, New York, Auerbach Publications, 2006. 392 p.
3. **Shalyto A. A.** *SWITCH-tehnologija. Algoritmizacija i programmirovanie zadach logicheskogo upravlenija* (Switch-technology. Algorithmization and programming of logic control problems), Saint-Petersburg, Nauka, 1998, available at: http://is.ifmo.ru/books/switch/1
4. **Machina.js,** available at: http://machina-js.org/
5. **Boost** Meta State Machine, available at: https://www.boost.org/doc/libs/1_67_0/libs/msm/doc/HTML/index.html
6. **UniMod**, available at: http://unimod.sourceforge.net/
7. **Rosmaro**, available at. https://rosmaro.js.org/
8. **YAKINDU** Statechart Tools, available at: https://www.itemis.com/en/yakindu/state-machine/
9. **Parondzhanov V. D.** *Uchis' pisat', chitat' i ponimat' algoritmy* (Learn to write, read and understand algorithms). Moscow: DMK Press, 2012, 520 p. (in Russian).
10. **DRAKON** Editor, available at: http://drakon-editor.sourceforge.net/
11. **DrakonHub,** available at: https://drakonhub.com/
12. **Mitkin S.** An interactive state machine demo, available at: https://drakonhub.com/files/lift.html
13. **Miro Samek,** Introduction to Hierarchical State Machines, 2016, available at: https://barrgroup.com/Embedded-Systems/How-To/Introduction-Hierarchical-State-Machines