4. Фисенко В. Т. Компьютерная обработка и распознавание изображений: учеб. пособие / В. Т. Фисенко, Т. Ю. Фисенко. – СПб.: СПбГУ ИТМО, 2008. – 192 с.

5. Видеостены и качество передачи изображения на составных полиэкранах [Электронный ресурс]. – Режим доступа: http://kljuch.ru/videosteny-s-kachestvo-peredachi-izobrazheniya-na-sostavnyh-poliekranah/. – Название с экрана. – Дата доступа 11.05.2011.

УДК 004.94:519.876

Р.Ю. Лопаткин, канд. физ.-мат. наук
В.А. Иващенко, аспирант
Институт прикладной физики НАН Украины, г. Сумы, Украина

ЯЗЫК ПРОГРАММИРОВАНИЯ GROOVY КАК СРЕДСТВО ОПИСАНИЯ МУЛЬТИАГЕНТНЫХ МОДЕЛЕЙ

В данной статье авторами предлагается использование методологии метапрограммирования для упрощения описания мультиагентных моделей, предложена архитектура среды для мультиагентного моделирования в рамках данного подхода, а также решаются связанные с этим подходом технические вопросы.

Введение

Перед разработчиками среды имитационного моделирования всегда стоит задача предоставления пользователю некоторого интерфейса для описания моделей. Обычно исследователю предоставляется возможность описывать модель либо с помощью специального графического интерфейса, либо используя языки программирования. Понятно, что первый подход может обеспечить значительное упрощение процесса описания моделей, а второй – его гибкость. Для совмещения преимуществ обоих подходов, с нашей точки зрения, напрашивается использование графических интерфейсов для задания параметров модели, а для описания модели – использовать язык программирования.

Описания моделей с помощью языка программирования возможно несколькими способами:

1. Интегрировать скриптовый язык в систему моделирования, написанную на компилируемом языке программирования. Примером такой связки могут быть язык программирования C++ и скриптовый язык Luna Lua или Angel Script. Преимуществом первого подхода может быть эффективное обеспечение безопасности – код скриптового языка часто (но не всегда) выполняется в специальной защищенной среде, за пределы которой скрипт не имеет доступа. Но такой подход может иметь недостатки относительно обеспечения взаимодействия среды моделирования с моделью, поскольку для полноценного взаимодействия необходимо, чтобы среда моделирования имела доступ к любой переменной, любой функции модели, а модель, в свою очередь, – наоборот, должна иметь доступ к переменным среды моделирования. Еще очевидный недостаток такого подхода – отсутствие кроссплатформенности разработанной программы, что является критическим фактором при разработке распределенных систем.

2. Описывать модели на том же языке, на котором разработана среда моделирования. Такой подход может обеспечить высокое быстродействие в случае, если идет речь о компилируемом языке программирования, но возникают вопросы с безопасным выполнением кода модели, а также с гибкостью интеграции модели и среды моделирования. Для решения вопросов безопасности в таком случае придётся создавать или использовать готовую виртуальную машину, что не всегда удобно и может збавить от преимущества в скорости выполнения. Также как и в первом случае не обеспечивается кроссплатформенность разработанного решения. Подобный подход может быть приемлем, если среда моделирования предназначена для запуска на локальной машине, но неприемлем при распределенном моделировании. Примером такой системы может

служить PSim [1]. Но совсем по-другому может обстоять вопрос, если среда разработана на интерпретируемом языке и на этом же языке описываются модели.

3. Альтернативой первым двум подходам может быть разработка специального скриптового языка, который изначально будет ориентирован для решения поставленной задачи. Однако это требует значительных трудозатрат и на его реализацию может уйти много времени. Такой подход реализован, например, в NetLogo [2].

Основная часть

Таким образом, каждый из этих подходов имеет свои преимущества и недостатки. Как мы считаем, использование платформы Java решает множество проблем, присущих рассмотренным подходам: она обеспечивает кроссплатформенность разработанных приложений, решает немаловажный вопрос безопасности выполнения кода моделей. Вопрос безопасности становится особо остро в случае создания распределенных систем моделирования, когда код модели может передаваться системе моделирования с удаленной машины. А, как известно, вопросы безопасности и ограничения доступа для кода можно решать с помощью стандартных средств платформы Java, которые сильно развиты [3]. Но, с другой стороны, при использовании платформы Java возникают некоторые проблемы с гибкостью созданных решений, избыточностью кода моделей не только в плане избыточности самого кода Java, но и в плане наличия в таком случае множества «шаблонных» конструкций, присущих моделям.

Но эти проблемы можно решать использованием современных языков программирования, ориентированных на платформу Java, реализующих новые концепции программирования не присущие языку Java. Поэтому с целью создания основы специализированного языка программирования для описания моделей мы использовали Groovy [4], как одну из самых удачных, на наш взгляд, реализаций такого языка, поддерживающего концепцию метапрограммирования.

На рисунке 1 схематически представлена, предлагаемая нами, архитектура среды мультиагентного моделирования, которая позволила бы использовать платформу Java в сочетании с Groovy, обеспечивая простую интеграцию и, кроме того, делать надстройки над языком программирования Groovy для решения, указанных выше, проблем.

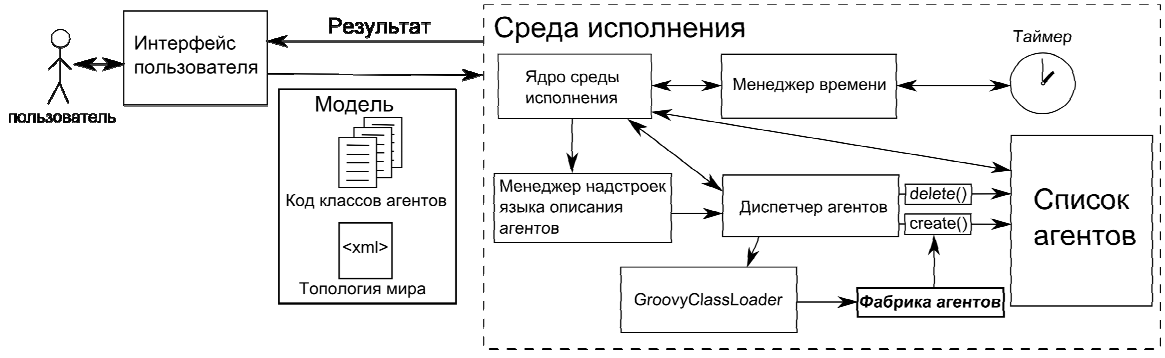

Рис. 1. Архитектура системы мультиагентного моделирования

Как видно из рисунка, пользователь с помощью *интерфейса пользователя* описывает модель и передает её далее *среде исполнения* модели в виде *кода классов агентов* и файла в формате XML, в котором задаются характеристики модельного мира. В самом начале *ядро среды исполнения* обращается к *менеджеру надстроек языка описания агентов* и с его помощью активирует надстройку Groovy, которая является набором языковых конструкций, позволяющих описывать логику агентов. Все надстройки, используемые *менеджером надстроек описания агентов,* хранятся в одном отдельном файле, поэтому простая замена этого файла обеспечивает поддержку новых конструкций, которые могут быть разработаны в рамках рассматриваемого подхода.

Все активные агенты создаются, хранятся и удаляются из *списка агентов* только *диспетчером агентов*, что происходит после команды *ядра среды исполнения*, которое осуществляет изменение состояний агентов во времени, а также взаимодействует с *менеджером времени*, отвечающим за продвижение модельного времени. Для создания новых агентов модельного мира *диспетчер агентов* использует *фабрику агентов*, которая создается с помощью класса *GroovyClassLoader*.

Покажем на примере, как можно использовать надстройки Groovy для упрощения описания моделей, избавления от часто используемых строчек кода. Рассмотрим в качестве примера известную модель «хищник-жертва». Описание класса агента-хищника может выглядеть следующим образом:

```
class Wolf extends AbstractAgent {
Random rand;
double x,y;
double min_x = 0.0;   //Эти переменные обязательны
double min_y = 0.0;   //и служат для ограничения
double max_x = 100.0; //мира, за пределы которого
double max_y = 100.0; //агенты не могут выходить
…
public Wolf() {
            rand = new Random();
            x = min_x+rand.nextDouble()*(max_x-min_x);
            y = min_y+rand.nextDouble()*(max_y-min_y);
 }
   …
}
```

Переменные `min_x`, `min_y`, `max_x`, `max_y` в данном случае используются как границы модельного мира, в котором могут существовать хищники, а x и y задают координаты агента в двухмерном пространстве. Как видно из конструктора класса, он создает агентов в рамках модельного мира с равномерным законом распределения координат.

Теперь, если рассмотреть «жизнедеятельность» агента, то становится понятно, что переменные `min_x`, `min_y`, `max_x`, `max_y` обязательны к использованию:

```
 // «Передвижение волка»
 int sign = Math.abs(rand.nextInt())%2;
 if(sign==1){
 x = x + delta;
 if(x>max_x) x = max_x;
 }
 else{
x = x - delta;
if(x<min_x) x = min_x;
 }

 sign = rand.nextInt()%2;
 if(sign==1){
y = y + delta;
if(y>max_y) y = max_y;
 }
 else{
```

```
y = y - delta;
if(y<min_y) y = min_y;
 }
```

Жирным курсивом выделен код, который приходится использовать в каждой подобной модели, но от него желательно было бы избавиться.

Теперь рассмотрим, как можно избавиться от такого кода с помощью метапрограммирования, например, с использованием языка Groovy. Во-первых, генерацию координат, создаваемых агентов, в таком случае можно осуществлять более простым и понятным образом:

```
class Wolf extends AbstractAgent {
def x = [0.0,100.0].random
def y = [0.0,100.0].random
…
}
```

Во-вторых, запретить выход чисел за пределы указанного диапазона можно на уровне самого языка и код в таком случае будет выглядеть следующим образом:

```
int sign = Math.abs(rand.nextInt())%2;
if(sign==1)
   x = x + delta;
else
   x = x - delta;

sign = Math.abs(rand.nextInt()%2);
if(sign==1)
   y = y + delta;
else
   y = y - delta;
```

Здесь гарантируется, что значение переменных `x` и `y` не выйдет за пределы отрезка `[0.0,100.0]`. Отметим, что такая запись возможна только при условии применения специальной надстройки над языком Groovy.

Рассмотрим подробно код этой надстройки. В примере происходит присваивание `def x = [0.0,100.0].random`, где конструкция `[0.0,100.0]` создает новый список из двух действительных чисел 0.0 и 100.0. Мы имеем возможность создать для этого списка метод получения случайного числа, заключенного между двумя этими числами, следующим образом:

```
ArrayList.metaClass{
 getRandom{->
def (mn,mx) = delegate
new Double(mn,mx)
       //возвращаем новую переменную типа Double
}
}
```

В данном коде происходит расширение функционала уже существующего класса `ArrayList`, представляющего собой список. В переменные `mn` и `mx` записываются значения концов отрезка, а затем происходит вызов конструктора стандартного типа `Double`. Расширение функциональности класса `Double` происходит следующим образом:

```
Double.metaClass{
def oldPlus = Double.metaClass.getMetaMethod("plus",
 [Integer] as Class[])
def oldMinus = Double.metaClass.getMetaMethod("minus",
 [Integer] as Class[])
double min
double max
Random Rand = new Random()
getMin={->min}
setMin={->min = delegate}
getMax={->max}
setMax={->max = delegate}
constructor={double mn, double mx->
min = mn
max = mx
return min+(max-min)*Rand.nextDouble()
//Записываем значение только что созданной переменной Double
 }
plus = {Double n->
def res = oldPlus.invoke(delegate,n)
if(res<=max)
return res;
else
return max;
}
minus = {Double n->
def res = oldMinus.invoke(delegate,n)
if(res>=min)
return res;
else
return min;
}
}
```

Теперь в классе `Double` появились дополнительные поля: `min` – допустимое минимальное значение числа, `max` – допустимое максимальное значение числа, методы для их получения и установки: `getMin`, `setMin`, `getMax`, `setMax`. Добавленный новый конструктор записывает границы допустимых значений числа и генерирует случайным образом значения для числа с учетом границ. А также происходит переопределение операций «+» и «–», которые уже не позволяют числу выходить за рамки его диапазона.

Как видно, оптимизация кода возможна не только потому, что код на Groovy менее избыточный, чем на Java, но и еще за счет возможности избавиться от шаблонных конструкций посредством метапрограммирования или заменить их форму записи на более удобную.

Необходимо отметить, что разработчики известной среды моделирования Repast Symphony [5] используют Groovy для описания моделей, но в данном программном продукте реализованы только базовые надстройки, которые нет возможности модифицировать до полноценного специализированного языка. Реализованная нами схема позволяет существенно упростить для пользователя процесс описания моделей. При этом для разработчиков открываются новые возможности создания плагинов в виде языковых конструкций, которые легко могут подключаться к разрабатываемой нами системе агентного моделирования.

Предложенный подход также позволяет просматривать и задавать параметры агентов с помощью графического интерфейса. Например, предположим, что у класса Wolf есть следующие свойства:

```
class Wolf extends AbstractAgent {
 def max_life = 6
 def life = max_life
 def eat_distance = [0.0,50.0].number
 def delta = [0.0,50.0].number
 def max_hungry = 4
 def hungry = 2
 …
}
```

Этому коду будет соответствовать графический интерфейс, который показан на рис. 2:

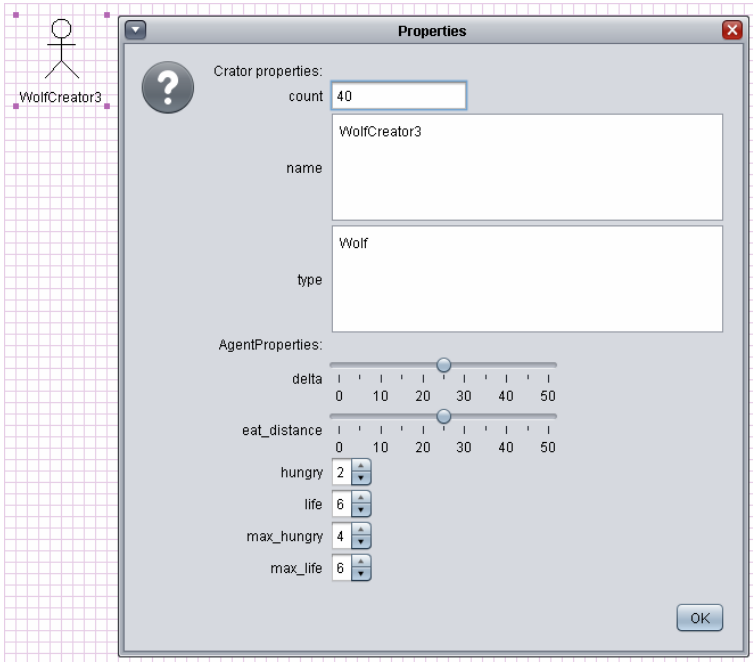

Рис. 2. Графический интерфейс пользователя для задания параметров агента

Получать или устанавливать свойства агента подобным образом можно с помощью технологии Groovy Beans [4].

Выводы

Предложенный нами подход, основанный на Groovy, в сочетании с предложенной архитектурой построения системы мультиагентного моделирования, позволяет, с одной стороны, использовать преимущества платформы Java, а с другой – устраняет некоторые недостатки стандартного языка программирования Java, которые касаются построения подобных систем и были перечислены выше. К преимуществам в данном случае можно отнести: полное разделение логики модели и интерфейсов управления, гибкость и кроссплатформенность создаваемых решений, безопасность выполнения моделей, что не всегда присуще другим кроссплатформенным решениям.

Список использованных источников

1. PSIM Software by Powersim Inc [Электронный ресурс]. – Режим доступа: http://www.powersimtech.com/. – Заголовок с экрана.

2. NetLogo [Электронный ресурс]. – Режим доступа: http://ccl.northwestern.edu/netlogo/. – Заголовок с экрана.

3. Java SE Security [Электронный ресурс]. – Режим доступа: http://www.oracle.com/technetwork/java/javase/tech/index-jsp-136007.html. – Заголовок с экрана.

4. Groovy - Home [Электронный ресурс]. – Режим доступа: http://groovy.codehaus.org/. – Заголовок с экрана.

5. C.M. Macal. Repast Simphony Runtime System, in C.M. Macal, M.J. North, and D. Sallach (eds.), Proceedings of the Agent 2005 Conference on Generative Social Processes, Models, and Mechanisms, ANL/DIS-06-1, cosponsored by Argonne National Laboratory and The University of Chicago, Oct. 13–15, 2005.

УДК 621.365.9:620.197.3

В.П. Войтенко, канд. техн. наук
А.А. Королев, канд. техн. наук
П.К. Комзол, студент
Черниговский государственный технологический университет, г. Чернигов, Украина

ИСПОЛЬЗОВАНИЕ ИМПУЛЬСНОГО ПРЕОБРАЗОВАТЕЛЯ С ЦИФРОВЫМ КВАЗИОПТИМАЛЬНЫМ РЕГУЛЯТОРОМ В ПОТЕНЦИОСТАТЕ

С учетом возможностей, предоставляемых современной элементной базой систем информационной и энергетической электроники, сформулированы требования к потенциостату для исследований в области электрохимической коррозии. Предложены функциональные схемы, а также алгоритмы работы как резидентной части прибора, так и прикладного программного обеспечения для персонального компьютера.

Введение

Потенциометрия представляет собой широко распространенный метод определения различных физико-химических величин, основанный на измерении электродвижущих сил сопряженных электрохимических процессов. Потенциометрия, в частности, находит применение при разработке эффективных ингибиторов коррозии [1; 2]. Исследования обычно проводят с использованием потенциостата, представляющего собой сложный и дорогостоящий прибор, позволяющий с высокой точностью поддерживать заданный потенциал или ток рабочего электрода, изменять их по требуемому закону, а также измерять и регистрировать потенциал рабочего электрода и тока поляризации [3; 4].

Современная элементная база электронных систем и достижения в области как силовой, так и информационной электроники позволяют разработать устройство, которое не только не будет уступать известным моделям, а наоборот, сможет существенно превзойти их по ряду параметров, предложить новые возможности и, в то же время, быть доступным как для исследовательских, так и для учебных лабораторий [5; 6].

Целью данной статьи является разработка концепции построения потенциостата нового поколения, способного существенно повысить эффективность потенциометрических экспериментов.

Формулирование требований к потенциостату

Очевидно, что проектируя потенциостат нового поколения, в качестве его базовых характеристик можно взять те, которые используются для оценки существующих приборов. Однако большинство количественных параметров, а также реализуемые функции решающим образом зависят от области применения прибора. Разумеется, можно создать универсальный потенциостат для всех требуемых на сегодняшний день применений, но такое решение будет наименее эффективным. Стоимость самого прибора, его эксплуатации и обслуживания, сложность работы с ним могут оказаться чрезмерными, и, в то же время, ряд функций останутся невостребованными. Поэтому более рациональным решением является подход, сочетающий в себе два начала: модульность и использование программируемой логики. И то, и другое позволяет рентабельно произво-