МИНИСТЕРСТВО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ, МЕХАНИКИ И ОПТИКИ

Факультет___**Информационных технологий и программирования**_____

Направление **Прикладная математика и информатика**_____Специализация _____
Математическое и программное обеспечение вычислительных машин.
Академическая степень ___**магистр математики**_____

Кафедра__**Компьютерных технологий**_____Группа_____6538_____

МАГИСТЕРСКАЯ ДИССЕРТАЦИЯ

на тему

ПРЕОБРАЗОВАНИЯ ПРОГРАММ, СОХРАНЯЮЩИЕ ПОВЕДЕНИЕ

Автор: А.П. ЛУКЬЯНОВА

Научный руководитель: Ф.А. НОВИКОВ

Санкт-Петербург
2007

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ

В настоящее время наблюдается некоторый разрыв между теоретическими основами информатики и практическим программированием. Строгие утверждения теории алгоритмов, при интерпретации в терминах парадигм программирования, фактически теряют практическую значимость и тонут под грузом многочисленных оговорок и ограничений, связанных с деталями и особенностями реальных систем программирования.

Одна из главных причин такого положения дел заключается, по нашему мнению, в том, что абстрактные модели, определяемые и исследуемые в теоретической информатике, имеют слишком высокий уровень абстракции и слишком далеки от современного практического программирования. Распространенные модели вычислимости в теории алгоритмов строятся как можно более компактно и опираются на минимальное количество простейших понятий, принимаемых как постулаты. Например, система команд машины Тьюринга предельно проста: чтение/запись символов, сдвиги головки чтения записи, изменение номера состояния. Это отвечает требованиям и традициям теоретических исследований. Но в практическом программировании ситуация прямо противоположная: чем сложнее и изощреннее административная система времени выполнения, чем богаче набор библиотек системы программирования, тем лучше. Машину Тьюринга нетрудно описать, но программировать на ней неудобно. Математически состоятельным способом описать *C++* пока не удалось, но тысячи людей программируют на этом языке.

Было бы очень заманчиво изобрести универсальную модель вычислимости, которая была бы, с одной стороны, достаточно близка к практическому программированию для того, чтобы на ее основе делать практически значимые выводы, а, с другой стороны, была бы достаточно формализована для того, чтобы к ней были применимы математические методы анализа. Такие попытки, более или менее успешные, предпринимались неоднократно. Достаточно вспомнить *Algol-68*, машины абстрактных состояний Гуревича, семантику неподвижной точки Скотта и многие другие подходы. Несмотря на видимую

полезность и плодотворность этих попыток, ни одна из них не получила всеобщего признания, как универсальная модель. Однако отсутствие панацеи не является поводом для отказа от лечения болезней современного программирования. Как правило, в таких случаях используется общеизвестный прием: пожертвовать претензиями на универсальность в пользу практической применимости для решения конкретных задач. Один из примеров – схемы Лаврова, позволившие исчерпывающим образом решить задачу глобальной экономии памяти и в то же время практически не применявшиеся в других ситуациях.

Предлагаемая работа принадлежит к последнему классу. Здесь исследуется один вопрос, долгое время служивший и до некоторой степени служащий и сейчас предметом дискуссий практикующих программистов – почему сосуществуют и даже все время появляются новые парадигмы (или стили) программирования? В чем общее и в чем различия этих парадигм? Существуют ли между ними непреодолимые барьеры или выделение парадигм программирования не более чем игра в (модные) слова? С точки зрения формальной теории алгоритмов это праздный вопрос: все парадигмы программирования вычислительно полны и с теоретической точки зрения эквиваленты. Но с практической точки зрения это не так. Ограничившись самым распространенным (и наиболее близким автору) императивным программированием (то есть программированием, в котором допускается присваивание, и, значит, используется понятие времени) [1–3], наблюдаем множество внешне различных парадигм, каждая из которых имеет свои достоинства и недостатки, своих апологетов и хулителей. Даже принимая тезис Черча, практики вряд согласятся признать все эти парадигмы неразличимыми.

Для конкретного рассмотрения здесь выбраны три характерных императивных парадигмы: процедурное программирование [4], как наиболее заслуженное, автоматное программирование [5–7], как излюбленное автором, и объектно-ориентированное [8, 9], как наиболее полярное. Сравнивать парадигмы в внешней точки зрения не имеет особого смысла, поскольку

результат такого сравнения заведомо окажется зависимым от изначальной выбранной внешней точки зрения и не будет иметь объективным. Гораздо интереснее построить представительные модели для избранных парадигм и исследовать вопрос, возможен ли переход (преобразование программы) из одной парадигмы в другую с сохранением сущностных характеристик, а если возможен, то чего стоит такой переход с вычислительной точки зрения?

В первой главе вводится понятие поведения (программы) как сущностной характеристики, сохранение которой при преобразованиях нас интересует, а также формулируется метод исследования. Во второй главе выбираются средства описания моделей (парадигм). В третьей главе исследуется преобразование процедурных программ в автоматные и обратно, формулируется основной алгоритм преобразования, и исследуются его свойства. В четвертой главе рассматривается и доказывается возможность преобразования объектно-ориентированных программ в автоматные и обратно. В заключении сформулированы полученные результаты.

1. ПОСТАНОВКА ЗАДАЧИ И МЕТОД ИССЛЕДОВАНИЯ

Пусть X – множество наборов символов алфавита входных воздействий (событий), а Z – множество наборов символов алфавита выходных воздействий (реакций). Назовем (детерминированным) *поведением* любую функцию $f: X \to Z$ [10].

Замечание 1. Если соответствие f не является функцией, то поведение недетерминировано и здесь не рассматривается. Прилагательное «детерминированное» подразумевается и для краткости систематически опускается.

Замечание 2. Всякий детерминированный алгоритм (определенный в любой модели вычислимости) задает детерминированное поведение. Вопрос о том, возможно ли задание детерминированного поведения неалгоритмическими средствами, здесь не рассматривается.

Настоящая работа использует следующий *метод* для сравнительного изучения парадигм императивного программирования. Метод базируется на описании *теоретико-множественной модели* императивной программы, предложенном ниже в данной главе. Каждая исследуемая парадигма программирования формализуется ее *абстрактной программной моделью*, описание которой содержит:

- описание *синтаксиса программ*. Абстрактный синтаксис программы парадигмы задается диаграммой классов (метамоделью). Он определяет структуру данных для описания поведения. Если нужно, дополнительные контекстные условия записываются в форме логических утверждений о свойствах классов метамодели. Описание поведения в какой-либо абстрактной программной модели в соответствии с заданным в ней синтаксисом и называется в работе *программой*;

- описание операционной семантики программ с помощью определения виртуальной машины (ВМ) (иначе говоря, интерпретатора или исполнителя), способной выполнять программы заданного синтаксиса,

удовлетворяющие заданным контекстным условиям. Описание ВМ включает систему команд (*Engine*), которые считаются выполнимыми примитивами и описание алгоритма интерпретации, который описывается методом раскрутки как программа для этой же виртуальной машины;

− доказательство императивности описанной абстрактной модели, то есть определение теоретико-множественной императивной модели в терминах описанной программной модели.

После того, как абстрактные модели для парадигм программирования определены, возможно изучение различных свойств этих моделей. В данной работе, в частности, исследуются возможности преобразования программ различных моделей.

Замечание 3. Все описанные программные модели являются абстрактными, то есть их реализация жестко не фиксируется. В частности, виртуальная машина описывает операционную семантику программы, а не реализацию интерпретатора на конкретном языке программирования.

Итак, в работе описаны три абстрактные программные модели, соответствующие наиболее популярным парадигмам императивного программирования: процедурная, автоматная и объектно-ориентированная (ОО). Модели построены так, чтобы быть простыми в описании, но при этом достаточно сложными для того, чтобы служить адекватным представителем выбранной парадигмы. Преимуществом описанных моделей служит возможность графического изображения программ: для описания программ в каждой из описанных моделей используются диаграммы определенного типа: диаграммы деятельности [11–17] (блок-схемы [18–20]) для процедурной модели, диаграммы состояний (графы переходов) [5] для автоматной, расширенные диаграммы классов [21] для объектно-ориентированной. Используются нотации диаграмм из стандарта *UML* [22, 23] с некоторыми ограничениями, выбранными таким образом, чтобы не ограничивать выразительную силу диаграмм.

Основной целью работы является исследование возможности преобразования описания поведения *f* (программы) в одной программной модели в описание того же самого поведения *f* в другой модели. Такие преобразования программ будем называть *сохраняющими поведение*.

Замечание 4. Если программа *B* получена из программы *A* преобразованием, сохраняющим поведение, то программа *A функционально эквивалентна* программе *B* в смысле теории вычислимости. Вопрос о том, возможна ли функциональная эквивалентность, не сохраняющая поведение, здесь не рассматривается.

1.1. Абстрагирование метода

Для абстрактного описания вычислительного состояния некоторого процесса или для произвольного набора данных в работе используется понятие *контекста*. Пусть контекстом называется область памяти вычислителя произвольного размера, хранящая поименованные переменные произвольного вида. Они могут быть типизированными или не типизированными, структура контекста может быть фиксирована или произвольна, данные в нем могут быть доступны только для чтения или записи, а могут допускать оба типа операций. Пусть контекст определяется с помощью *описателя контекста*, который задает свойства контекста, например, список имен переменных и\или их типы, возможность модификации и т.п. Примерами программной реализации контекста могут служить записи в языке *Pascal* (типизированный, фиксированный набор полей) или *java.util.HashMap* в языке *Java* (не типизированный, произвольный набор). Пусть множество всех вычислительных состояний контекстов всех типов и конечных размеров обозначается как C^*. Тогда элемент множества C^* есть набор данных произвольного вида, отсюда следует, что определение понятия *поведение* переписывается следующим образом: поведением является любая функция *f* вида:

$f: X \to Z,$ где $X,Z \subset C^*$.

Заметим, что императивная парадигма предлагает реализовывать поведение в виде набора команд, среди которых есть *команды управления* и *команды вычисления*. Предлагается считать команды вычисления также поведениями, которые могут быть элементарными и неделимыми инструкциями вычислителя или в свою очередь описываться некоторой императивной программой. Предполагается по определению, что виртуальная машина (вычислитель) каждой описанной в работе программной модели может выполнять некоторый (для всех моделей общий) набор встроенных вычислительных команд, а также вызывать императивные поведения, реализованные в согласии с программным синтаксисом модели.

Замечание 5. Способы реализации контекстов и их описателей, а также элементарных инструкций ВМ, в конкретном языке программирования могут быть различны, и этот вопрос выходит за пределы данного исследования, поскольку не влияет на его выводы.

Введенные понятия позволяют сделать все дальнейшие описания абстрактными, в то же время оставляя их строго формализованными.

1.2. Теоретико-множественное описание императивной парадигмы

Императивное программирование характеризуется принципом последовательного изменения состояния вычислителя пошаговым образом. Исходя из этого принципа, описание поведения в императивной форме будет итеративным, то есть функция $f: X \to Z$ будет представима в виде рекурсивного применения (композиции) некоторой другой функции или набора функций. При этом состояние процесса на некотором шаге будет описываться парой: его управляющим состоянием и вычислительным состоянием. Тогда любое императивное поведение описывается как восьмикомпонентный кортеж:

$$P = (S \times Y, X, Z, \mu, \lambda, \nu, s_{нач}, S_{закл}),$$

где S – множество управляющих состояний (каждому состоянию соответствует вычислительная команда программы), Y – множество вычислительных состояний системы, X – множество возможных входных наборов, Z –

множество выходных наборов, λ – функция переходов (к следующей команде), v – функция вычислений (модификации вычислительного состояния системы), μ – функция выходов, формирующая выходной набор из множества Z, $s_{нач}$ – начальное управляющее состояние, $S_{закл}$ – набор заключительных управляющих состояний, причем:

$\mu : X \times S \times Y \to Z;$

$v : X \times S \times Y \to Y;$

$\lambda : X \times S \times Y \to S.$

Замечание 6. Три описанные функции задают общую функцию изменения состояния: $\xi: X \times S \times Y \to S \times Y \times Z$ следующим образом:

$\xi(x, s, y) = (s', y', z') \Leftrightarrow \mu(x, s, y) = z'$ **and** $v(x, s, y) = y'$ **and** $\lambda(x, s, y) = s'$.

Композиция функции ξ или рекурсивное применение трех функций μ, v и λ, стартующее с позиции $s = s_{нач}$, $y = \varnothing$, и завершающееся при условии $s \in S_{закл}$, определяет императивным способом поведение $f: X \to Z$:

$f(x) = z \Leftrightarrow y = \varnothing,\ s := s_{нач}$

while $s \notin S_{закл}$ *do*

$z := \mu\ (x, s, y)$

$y := v\ (x, s, y)$

$s := \lambda\ (x, s, y)$

end while

2. МОДЕЛИ ТРЕХ ПАРАДИГМ ПРОГРАММИРОВАНИЯ

Покажем теперь, как описанная теоретико-множественная модель императивного программирования реализуется с помощью программной модели в рамках трех рассматриваемых парадигм – автоматной, процедурной и объектно-ориентированной.

2.1. Автоматная модель

Будем рассматривать в данной работе класс детерминированных автоматов Мили [5] с памятью произвольного (контекстного) вида.

2.1.1. Абстрактный синтаксис автоматной программы

Будем определять автомат с помощью *диаграммы состояний StateMachine*. На рис. 1 представлена схема абстрактного синтаксиса диаграммы состояний автомата, основанная на синтаксисе диаграмм состояний стандарта *UML*.

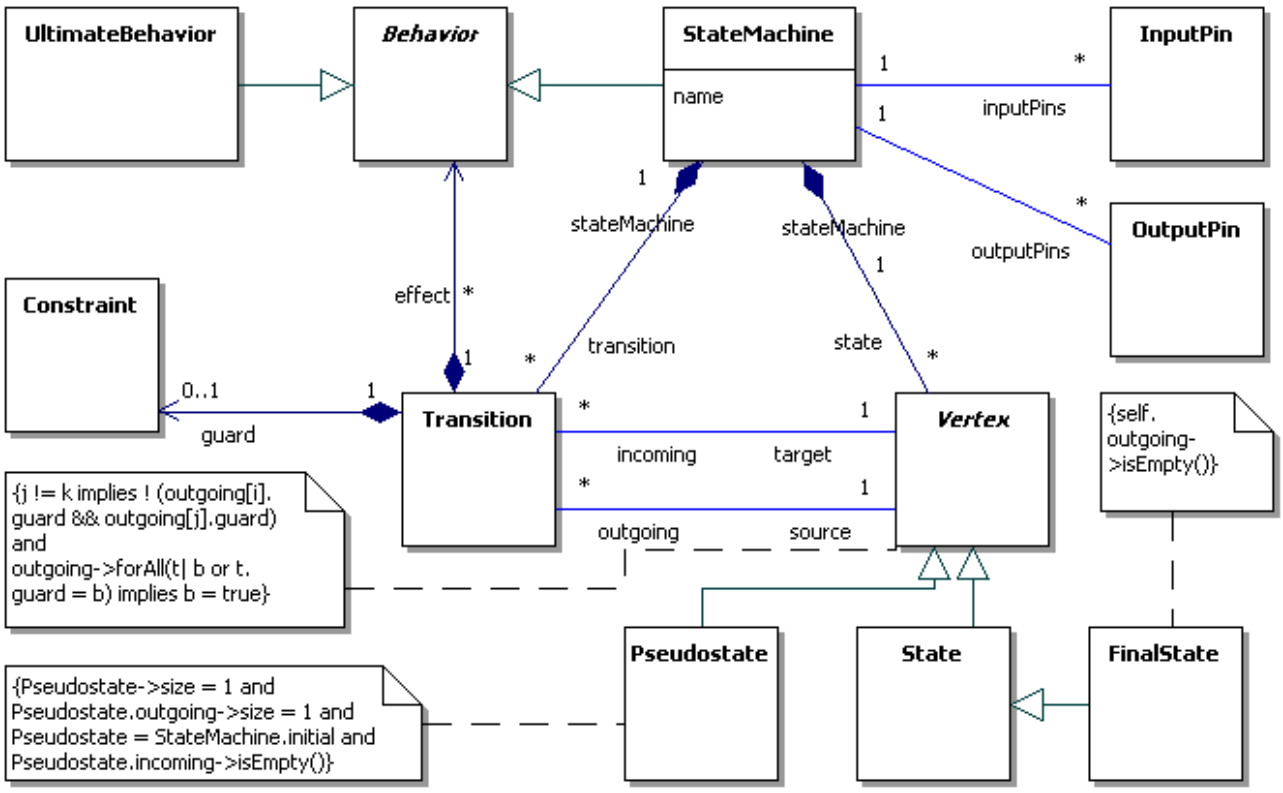

Рис 1. Абстрактный синтаксис диаграммы состояний

Одним из отличий описываемого синтаксиса от стандарта *UML* является понятие тип поведения *UltimateBehavior*. Поведения этого типа – это вычислительные команды (определены предыдущей главе), реализуемые ВМ

всех описанных в работе моделей. Они изменяют вычислительное состояние. Например, такой командой можно считать вызов оператора присваивания с вычислением арифметического выражения.

2.1.2. Виртуальная машина автоматной модели

Назовем виртуальную машину автоматной модели *AEngine* (рис. 2).

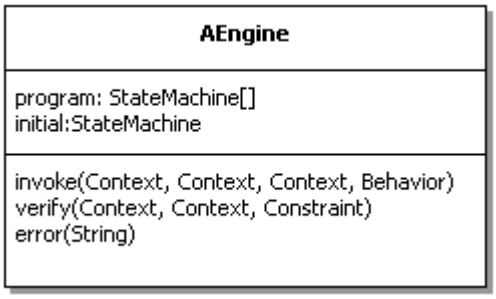

Рис 2. Виртуальная машина *AEngine* автоматной модели

Программа описывается как набор поименованных диаграмм *StateMachine*, дополненный именем головной диаграммы. При запуске программы виртуальная машина загружает в память набор описаний *StateMachine* и запускает головной автомат, передавая ему входные данные программы.

Покажем, как виртуальная машина реализует запуск автомата *StateMachine*. При вызове автомата во время выполнения программы (или при ее запуске) по диаграммному описанию автомата создается экземпляр автомата *Automaton* (рис. 3). Экземпляр автомата хранит свое текущее состояние *state*, ссылку на свое диаграммное описание и три контекста данных – входной, выходной и рабочий. Входной контекст описывается набором элементов *InputPin* синтаксиса программы и его данные доступны только для чтения. Входной контекст описывается набором элементов *OutputPin* и его данные доступны только для записи.

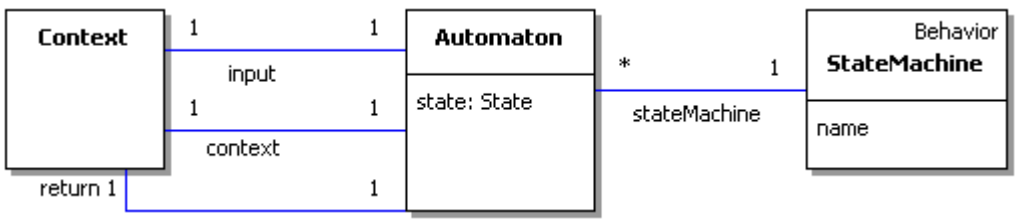

Рис 3. Автоматная программа во время выполнения

В числе методов виртуальной машины:

− метод *invoke(Context input, Context context, Context return, Behavior behavior)* реализует вызов вычислительной команды *UltimateBehavior* или автомата *StateMachine*;

− метод *verify(Context input, Context context, Constraint constraint)* реализует проверку условия *Constraint*.

Первый метод виртуальной машины принимает параметром контексты – входной, рабочий и возвратный – вызывающего экземпляра автомата. Из них специальным образом, зависящим от реализации модели, выделяются входные данные для вызова поведения *behavior*, и в них записываются выходные данные, полученные после вызова.

Второй метод ВМ не меняет вычислительное состояние, а только анализирует условие *Constraint*, используя данные из переданных ему параметрами контекстов.

Замечание 7. Способ формирования контекстов данных по предлагаемым контекстным описателям *InputPin*, *OutputPin*, не фиксируется абстрактной моделью. Однако предполагается, что он одинаков для всех моделей, использующих эти контекстные описатели.

Замечание 8. Все описанные в работе абстрактные ВМ при реализации будут использовать ряд дополнительных методов не только для работы с данными, но и, например, для построения абстрактных связей, аналогичных некоторым ассоциациям на диаграммах синтаксиса программы. Например, связь *Transition.effect* может быть описана именем вызываемого поведения, а ВМ может использовать словарь поведений для поиска нужного. Предполагается, что способы построения ассоциаций, также как и контекстов данных, реализуются аналогичным образом во всех виртуальных машинах, описанных в работе.

Как уже говорилось, реализация обработки вычислительных команд (*UltimateBehavior*) виртуальной машины (методом *invoke*) выходит за рамки

абстрактной модели. Опишем тогда реализацию метода invoke при вызове поведения, имеющего автоматное описание (*behavior.class = StateMachine*):

− виртуальная машина создает экземпляр *Automaton automaton* указанного типа *StateMachine*;

− входной контекст автомата *automaton.input* формируется в соответствии с описателем контекста *StateMachine.inputPins* и заполняется данными из контекстов *input* и *context*;

− выходной контекст *automaton.output* формируется в соответствии с его описателем *StateMachine.outputPins*;

− созданный экземпляр автомата запускается некоторым внутренним методом, абстрактную реализацию которого можно представить в виде диаграммы состояний, изображенной на рис. 4;

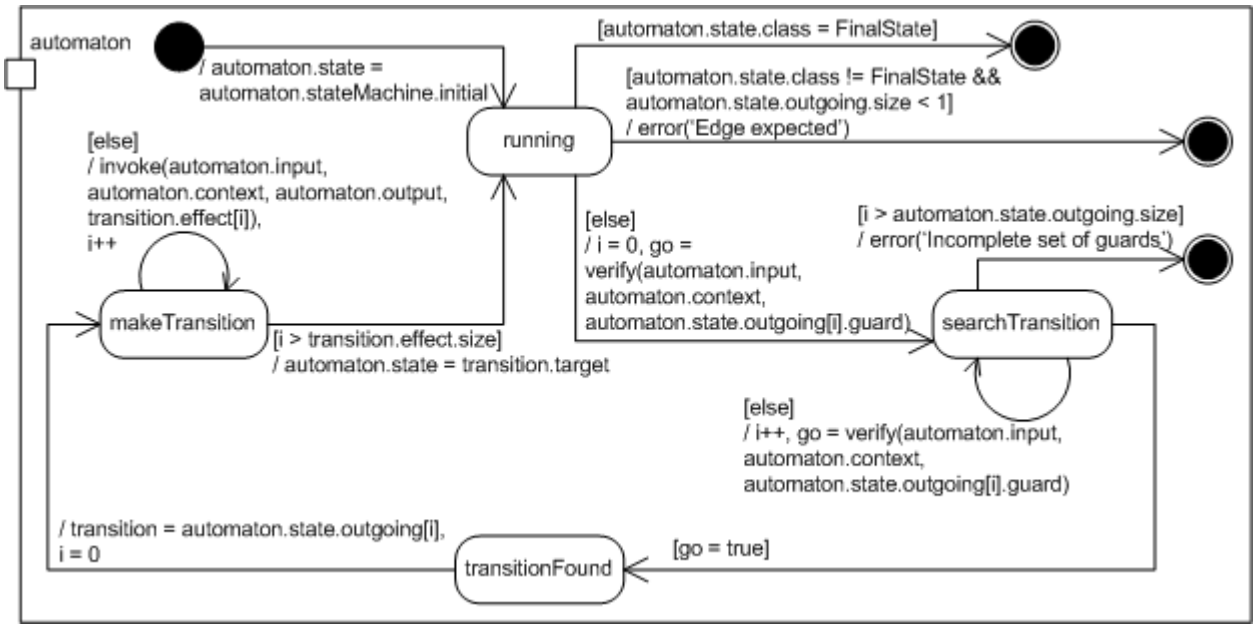

Рис 4. Автоматное описание метода *AEngine.invoke(automaton)*

− после прихода автомата в заключительное состояние данные его контекста возврата сохраняются в контекстах вызывающего автомата некоторым образом, зависящим от реализации ВМ.

2.1.3. Пример автоматной программы

На рис. 5 приведен простейший пример диаграммы состояний автомата Мили для программы поиска элемента в массиве.

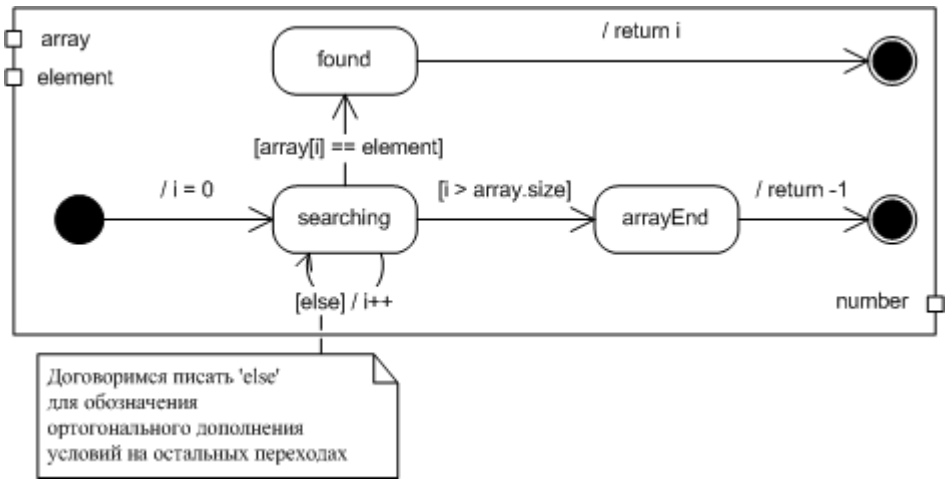

Рис 5. Пример диаграммы состояний: *number search(array, element)*

Входными данными программы является пара *(array, element)*. Счетчик цикла *i* хранится в контексте автомата. Возвращаемый результат *number* – номер первого появления элемента *element* в массиве *array* или *–1* если такого элемента в массиве нет. Для того чтобы не перегружать диаграмму, здесь и далее в примерах программ будем опускать обращение к конкретному контексту данных если оно очевидно, например, на данном рисунке вместо *array* следует читать *input.array*, а вместо *i* – *context.i*. Сохранение в выходной контекст будем обозначать словом *return* («*return i*» := «*return.number = i*»).

2.1.4. Императивность автоматной модели

Покажем, что программа, имеющая заданный синтаксис и интерпретируемая описанной абстрактной виртуальной машиной, является императивной, то есть определяет функциональную модель, описанную в предыдущей главе. Для этого покажем, как описанная программная модель (*StateMachine*) определяет императивную систему $P = (Y \times S, X, Z, \mu, \lambda, \nu, s_{нач}, S_{закл})$.

Заметим, что модель допускает два типа команд, вызываемых при интерпретации автомата, и эти команды (являющиеся поведениями) можно записать как функции следующего вида:

− $behavior: X \times Y \rightarrow Y \times Z$ – команда вызова поведения *Behavior* на переходе *Transition.effect*, реализуемая методом ВМ *invoke(Context input [readonly], Context context [read\update], Context return [updateonly],*

Behavior behavior), например, оператор присваивания «*i*++» или выражение «*return i*» в примере, изображенном на рис. 5;

− *constraint: X×Y → {0,1}* – команда проверки условия *Constraint*, реализованная методом *verify(Context input [readonly], Context context [readonly], Constraint constraint)*.

Тогда описанная программная модель (*StateMachine*) определяет императивную систему $P = (Y \times S, X, Z, \mu, \lambda, \nu, s_{нач}, S_{закл})$, где:

− *S := {? extends Vertex}* – множество вершин *State* или *Pseudostate* диаграммы *Automaton.stateMachine* является множеством управляющих состояний;

− $s_{нач}$ *:= Pseudostate* – начальное состояние системы;

− $S_{закл}$ *:= {FinalState}* – набор заключительных состояний;

− *Y := {вычислит. состояний контекста произвольного вида (Automaton.context)}*;

− *X := {вычислит. состояний контекста, описанного набором InputPin (Automaton.input)}*;

− *Z := {вычислит. состояний контекста, описанного набором OutputPin (Automaton.return)}*;

− функция выходов μ определяется следующим образом:

$$\mu(x, s, y) = z \Leftrightarrow \exists\, Transition\ t:$$
$$t.source = s\ \textbf{and}$$
$$[t.guard = null\ \textbf{or}\ t.guard(x,\ y)]\ \textbf{and}$$
$$[\exists z_i \in Z, \exists y_i \in Y: (y_{i+1}, z_{i+1}) = t.effect_i(x,\ y_i),$$
$$i=0,1...t.effect.size-1,\ y_0 = y$$
$$\Rightarrow\ z=\cup z_i].$$

Заметим, что в данном описании *t.guard(x, y)* подразумевает поведение *constraint: X×Y → {0,1},* реализуемое вызовом метода ВМ *verify*, $t.effect_i$ – поведение *behavior: X×Y→ Y×Z*, реализуемое методом ВМ *invoke*;

- функция переходов λ определяется следующим образом:

$$\lambda(x, s, y) = s' \Leftrightarrow \exists\, Transition\ t:$$
$$t.source = s\ \textbf{and}$$
$$[t.guard = null\ \textbf{or}\ t.guard(x,\ y)]\ \textbf{and}$$
$$t.target = s';$$

- функция вычислений ν определяется следующим образом:

$$v(x, s, y) = y' \Leftrightarrow \exists\, Transition\ t:$$
$$t.source = s\ \textbf{and}$$
$$[t.guard = null\ \textbf{or}\ t.guard(x,\ y)]\ \textbf{and}$$
$$[\exists z_i \in Z, \exists y_i \in Y: (y_{i+1}, z_{i+1}) = t.effect_i(x,\ y_i),$$
$$i=0,1...t.effect.size-1,\ y_0 = y$$
$$\Rightarrow\ y' = y_{t.effect.size}]$$

Замечание 9. О вложенных вызовах автоматов и рекурсии. Они реализуются как часть описанной теоретико-множественной модели императивной программы, так как μ, λ и ν в ней являются функциями, то есть по определению поведением, то есть могут быть реализованы, в том числе, вложенным автоматом.

2.2. Процедурная модель

Будем использовать нотацию *диаграмм деятельности UML* (*блок-схем*) для описания программы в процедурной модели.

2.2.1. Абстрактный синтаксис процедурной программы

Будем определять процедуру с помощью *диаграммы деятельности Activity*. На рис. 6 представлена схема абстрактного синтаксиса диаграммы.

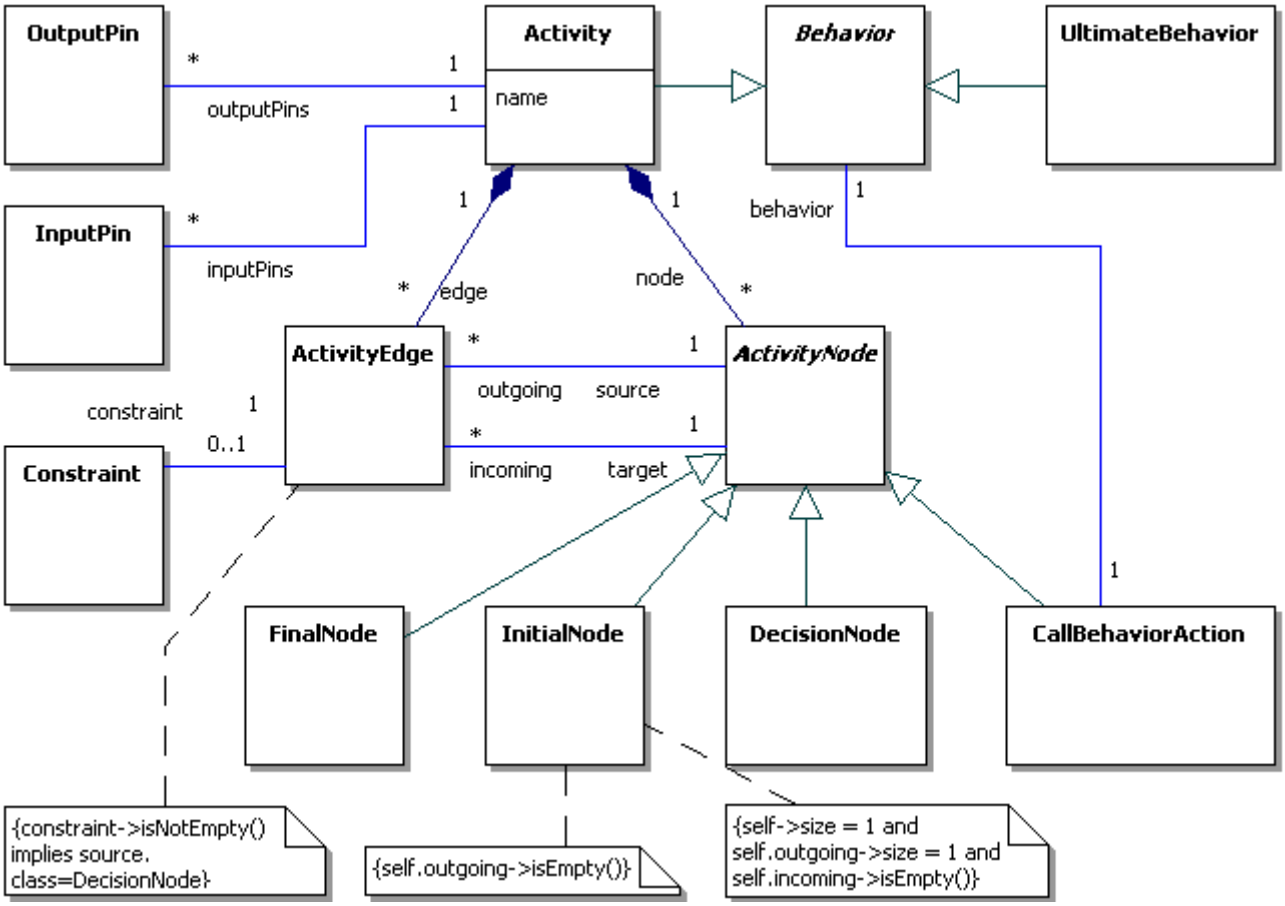

Рис 6. Абстрактный синтаксис диаграммы деятельности

2.2.2. Виртуальная машина процедурной модели

Назовем виртуальную машину процедурной модели *PEngine* (рис. 7).

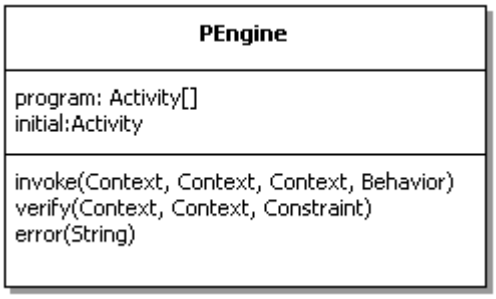

Рис 7. Виртуальная машина PEngine процедурной модели

Программа описывается как набор поименованных диаграмм *Activity*, дополненный именем головной диаграммы. При запуске программы виртуальная машина загружает в память набор описаний *Activity* и запускает головную, передавая ей входные данные программы.

Покажем, как виртуальная машина реализует запуск процедуры *StateMachine*. Во время выполнения программы при каждом вызове процедуры создается экземпляр процедуры *Procedure* для ее запуска (рис. 8). Процедура хранит три контекста – входной, выходной и рабочий – и ссылку на свое диаграммное описание.

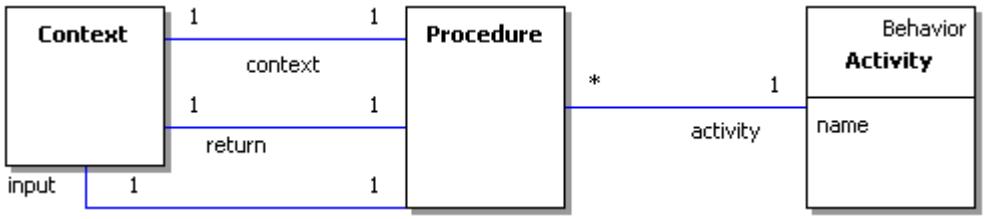

Рис 8. Процедурная программа во время выполнения

В числе методов виртуальной машины:

- метод *invoke(Context input, Context context, Context return, Behavior behavior)* реализует вызов вычислительной команды *UltimateBehavior* или процедуры *Activity*;
- метод *verify(Context input, Context context, Constraint constraint)*.

Первый метод виртуальной машины принимают параметрами контексты вызывающей процедуры. Из них специальным образом, зависящим от реализации модели, выделяются входные данные для передачи в вызываемое поведение и в них записываются выходные данные. Второй метод виртуальной машины реализует проверку условий *Constraint*, используя переданные параметрами контексты только для чтения данных.

Покажем реализацию вызова процедуры виртуальной машиной *PEngine* (рис. 9). Троеточие на этом рисунке заменяет получение контекстов процедуры с целью сделать диаграмму компактнее.

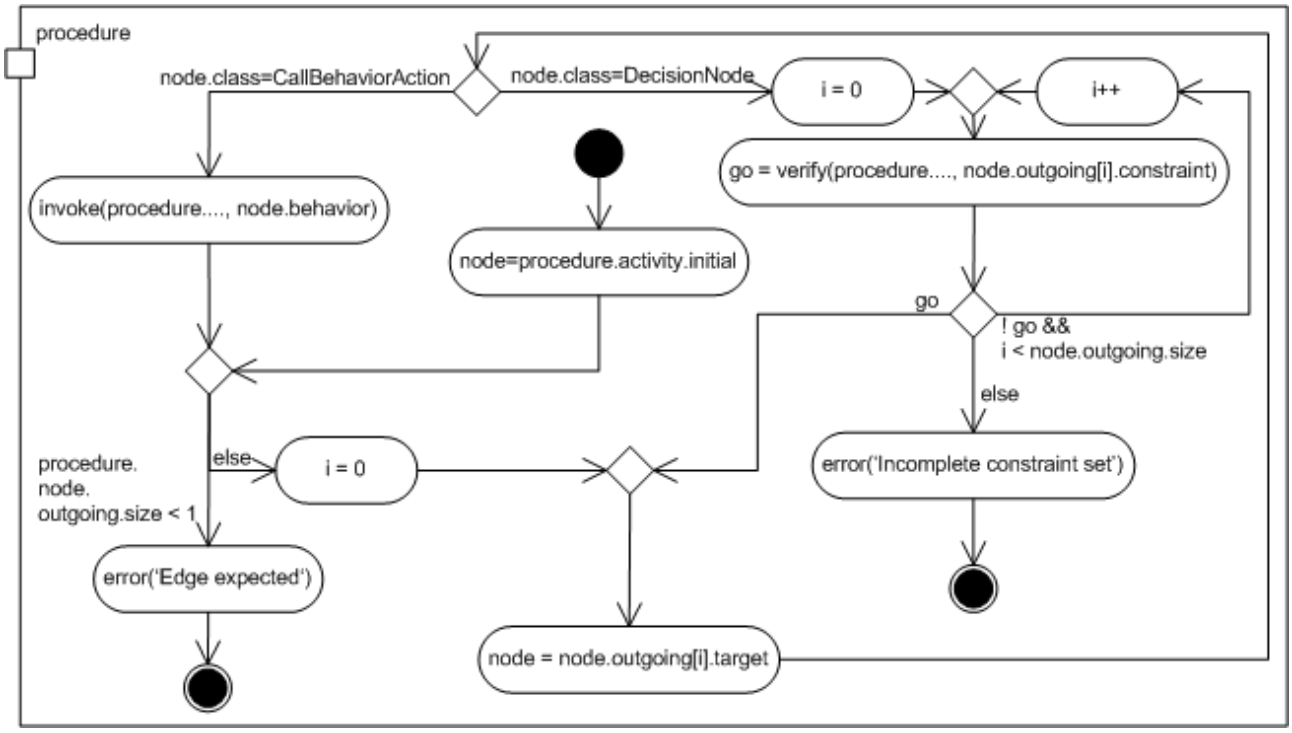

Рис 9. Описание метода *PEngine.invoke()* диаграммой деятельности

Отметим, что использованные в этой модели описатели контекстов *InputPin* и *OutputPin*, условия *Constraint*, и набор вычислительных команд *UltimateBehavior*, являются общей частью это и автоматной модели, а реализация методов ВМ для их обработки должна быть аналогична реализации ВМ автоматной модели (по условию метода, использованного в исследовании). Вследствие этого подробное их описание для процедурной модели не приводится. Единственное существенное отличие методов этой ВМ от методов автоматной – возможность вызова процедуры методом *invoke* – описана выше.

2.2.3. Пример процедурной программы

На рис. 10 приведен простейший пример диаграммы деятельности для программы поиска элемента в массиве.

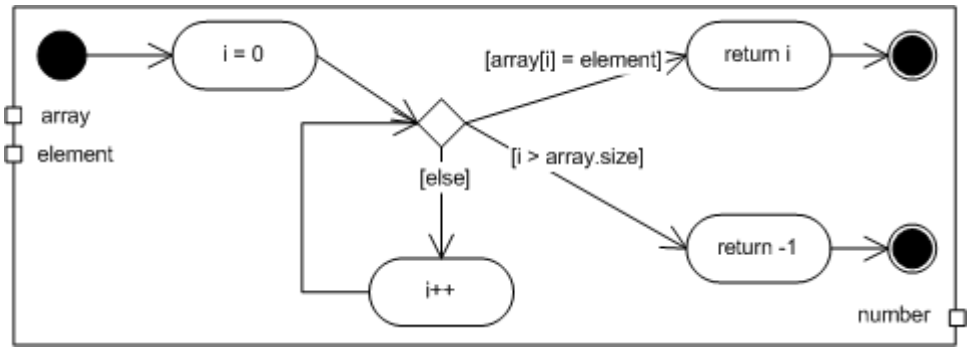

Рис 10. Пример диаграммы деятельности: number search(array, element)

Входными данными программы является пара *(array, element)*. Счетчик цикла *i* хранится в контексте процедуры. Возвращаемый результат *number* – номер первого появления элемента *element* в массиве *array* или *–1* если такого элемента в массиве нет.

2.2.4. Императивность процедурной модели

Покажем, что программа, имеющая заданный синтаксис и интерпретируемая описанной абстрактной виртуальной машиной, является императивной. Описанная программная модель (*Activity*) определяет императивную систему $P = (S \times Y, X, Z, \mu, \lambda, \nu, s_{нач}, S_{закл})$, где:

- $S := \{? \text{ extends } ActivityNode\}$ – множество вершин *ActivityNode* диаграммы деятельности является множеством управляющих состояний;
- $s_{нач} := InitialNode$ – начальное управляющее состояние;
- $S_{закл} := \{FinalNode\}$ – набор заключительных состояний;
- $Y :=$ *{вычислит. состояний контекста произвольного вида (Procedure.context)}*;
- $X :=$ *{вычислит. состояний контекста, описанного {InputPin} (Procedure.input)}*;
- $Z :=$ *{вычислит. состояний контекста, описанного {OutputPin} (Procedure.return)}*;
- функция выходов μ определяется следующим образом:

$$\mu(x, s, y) = z \Leftrightarrow [\ \exists\, CallBehaviorAction\ n, \exists y \in Y :$$

$$n = s \textbf{ and}$$

$(y', z) = n.behavior(x, y)\ ]$

or $[\ \exists DecisionNode\ n$ **:** $n = s$

and $z = \varnothing\ ]$

or $[\ \exists InitialNode\ n$ **:** $n = s$

and $z = \varnothing\ ]$.

Заметим, что в данном описании *n.behavior(x, y)* подразумевает поведение *behavior:* $X \times Y \rightarrow Y \times Z$, реализуемое методом ВМ *invoke*;

- функция переходов λ определяется следующим образом:

$\lambda(x, s, y) = s' \Leftrightarrow \exists ActivityEdge\ e$ **:**

$e.source = s$ **and**

$[e.guard = null$ **or** $e.guard(x, y)]$ **and**

$e.target = s'$.

Заметим, что в данном описании *e.guard (x, y)* подразумевает поведение *constraint:* $X \times Y \rightarrow \{0, 1\}$, реализуемое методом ВМ *verify*;

- функция вычислений *v* определяется следующим образом:

$v(x, s, y) = y' \Leftrightarrow [\ \exists CallBehaviorAction\ n, \exists z \in Z$ **:**

$n = s$ **and**

$(y', z) = n.behavior(x, y)]$

or $[\ \exists DecisionNode\ n$**:** $n = s$

and $y' = y\ ]$

or $[\ \exists InitialNode\ n$**:** $n = s$

and $y' = y\ ]$.

2.3. Объектно-ориентированная модель

Будем использовать нотацию *расширенных диаграмм классов* – композицию диаграммы классов и диаграммы деятельности, основанных на

нотации *UML* – для описания программы в объектно-ориентированной (ОО) модели.

2.3.1. Абстрактный синтаксис ОО программы

Программа в этой модели состоит из набора классов, который описывается диаграммой, имеющей синтаксис, изображенный на рис. 11.

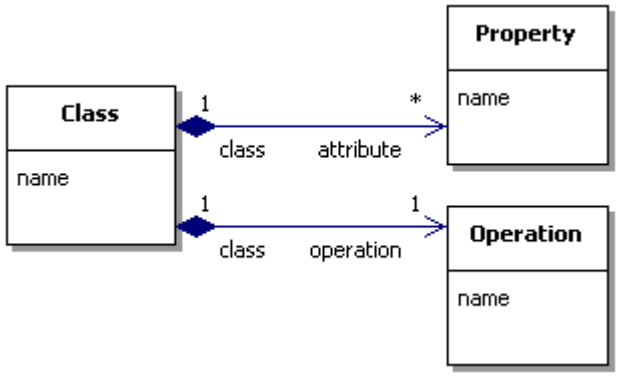

Рис 11. Общий абстрактный синтаксис ОО модели

Каждый метод *Operation* каждого класса задается диаграммами, схожими с диаграммами деятельности (рис. 12).

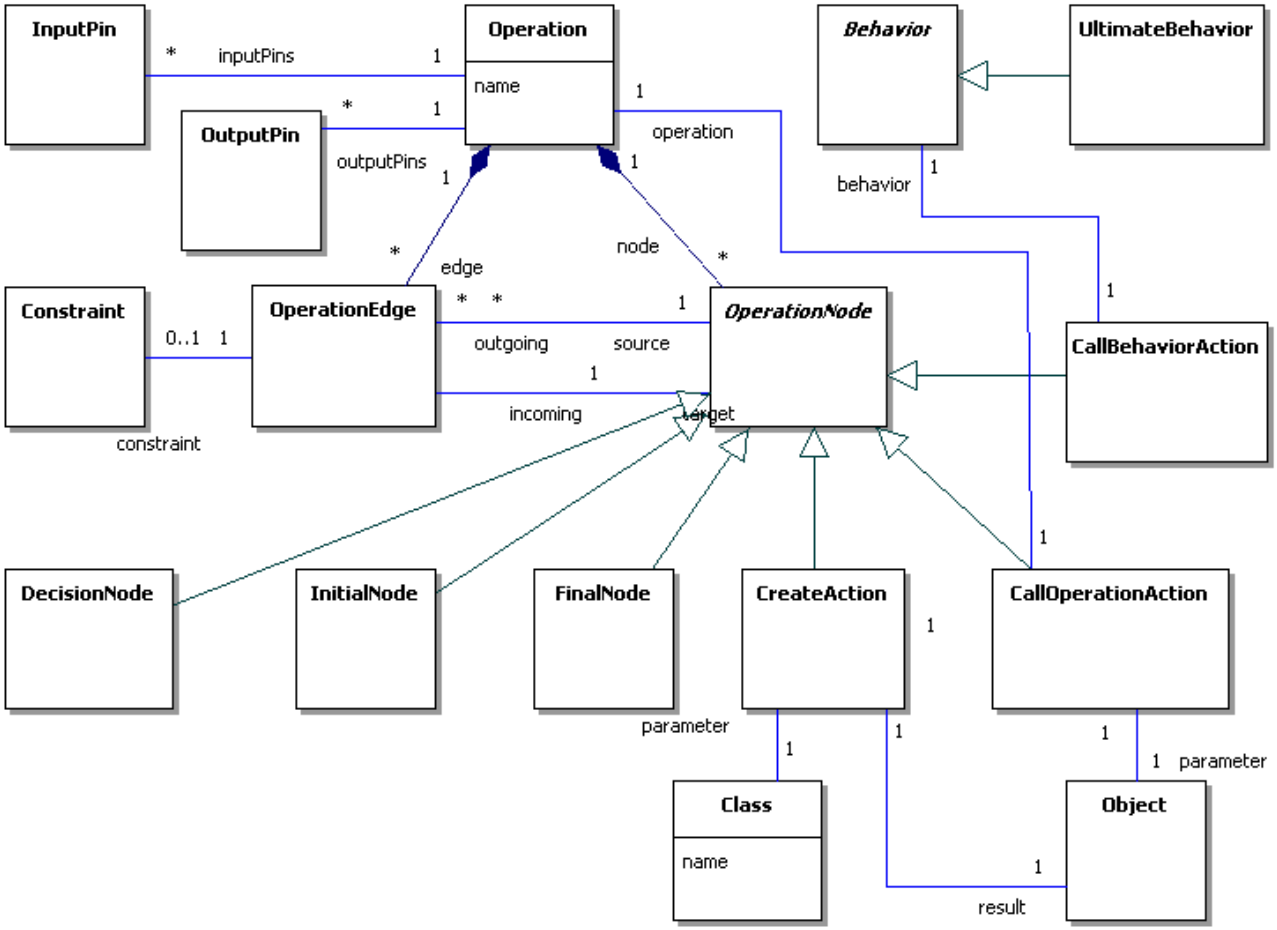

Рис 12. Абстрактный синтаксис метода

На рисунке опущены контекстные условия для описания метода, которые принимаются полностью аналогичными условиям процедурной модели.

2.3.2. Виртуальная машина ОО модели

Назовем виртуальную машину ОО модели *OEngine* (рис. 13).

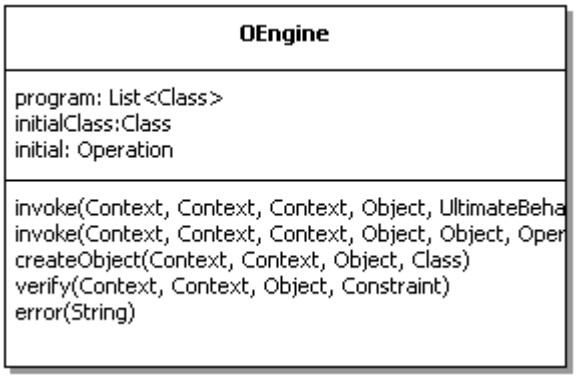

Рис 13. Виртуальная машина *OEngine* ОО модели

Во время выполнения программы для каждого класса может быть создано произвольное число экземпляров *Object*, называемых *объектами*. При каждом

вызове метода на указанном объекте создается экземпляр *Method* вызываемого метода (рис. 14).

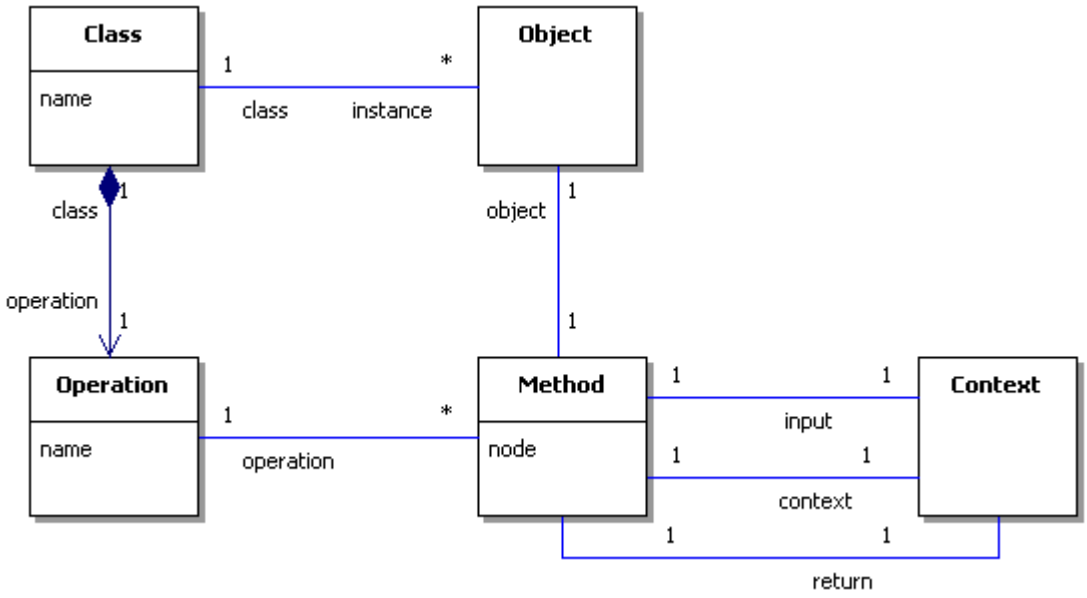

Рис 14. Программа во время выполнения

Объект *Method*, аналогично объекту *Procedure* в процедурной модели, хранит три контекста – входной, выходной и рабочий – и ссылку на свое диаграммное описание *Operation*. Но дополнительно к этому, он хранит еще и ссылку на объект, на котором метод был вызван.

Замечание 10. Элемент *Object* является элементом реализации ВМ (частью вычислительного состояния программы во времени выполнения) и, следовательно, не может присутствовать в явном виде при описании программы в соответствии с заданным синтаксисом. При описании программы вместо объекта *Object* подразумевается его уникальный идентификатор, расшифровка которого во время выполнения будет одним из примеров уже упоминавшихся скрытых команд ВМ, зависящих от реализации и не фиксируемых абстрактной моделью.

Программа описывается как диаграмма классов, дополненная именами головного класса и его метода. При запуске программы виртуальная машина создает экземпляр *Object* указанного класса, экземпляр *Method* его метода и запустит в нем указанную операцию.

В числе методов виртуальной машины:

− метод *invoke(Context input, Context context, Context return, Object object, UltimateBehavior behavior)*, реализующий вызов вычислительной команды. Его реализация отличается от реализации вызова вычислительной команды ВМ процедурной модели только тем, что ему доступен для чтения и модификации объект *object*, в дополнение к контекстам метода;

− метод *invoke(Context input, Context context, Context return, Object object, Object parameter, Operation operation)*, реализующий вызов метода *operation* объекта *parameter* с получением его входных данных из одного из контекстов вызывающего метода или из его объекта *object*;

− метод *createObject(Context context, Context return, Object object, Class class)*, создающий объект заданного класса *class* и сохраняющий его в рабочем или выходном контексте или как поле объекта *object*;

− проверка условия *verify(Context input, Context context, Object object, Constraint constraint)*, аналогичная методам *verify* описанных выше моделей, но имеющая доступ не только к данным входного и рабочего контекстов, но и к полям объекта *object*.

2.3.3. Пример ОО программы

На рис. 15 приведен простейший пример класса *ArrayList*, реализующего хранение списковых данных и имеющего пару методов – *search* (поиск) и *sort* (сортировка). Реализация методов аналогична примерам процедур *sort* и *search*, описанным в данной работе. Различием является то, что списковые данные хранятся в объекте, а не передаются как параметр процедуры.

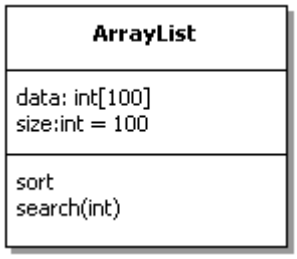

Рис 15. Пример класса – список *ArrayList*

2.3.4. Императивность ОО модели

Покажем, что программа, имеющая заданный синтаксис и интерпретируемая описанной абстрактной виртуальной машиной, является императивной. Рассматривая ОО программу как набор поведений (методов модели), имеющих доступ к некоторым внешним данным, хранимым в контексте *Object* с описателем *Class.attribute*, можно представить каждое поведение как императивное в теоретико-множественной модели, для которого вычислительное состояние состоит из пары элементов.

Метод модели *Operation* определяет императивную систему $P = (S \times Y \times O, X, Z, \mu, \lambda, \nu, s_{нач}, S_{закл})$, где

- S := *{? extends OperationNode}* – множество управляющих состояний;
- $s_{нач}$:= *InitialNode* – начальное управляющее состояние;
- $S_{закл}$:= *{FinalNode}* – набор заключительных состояний;
- Y := *{вычислит. состояний контекста произвольного вида (Method.context)}*;
- O = *{вычислит. состояний контекста, описанного Operation.class.attribute (Method.object)}* – контекст объекта;
- X = *{вычислит. состояний контекста описанного набором {InputPin} (Method.input)}*;
- Z = *{вычислит. состояний контекста, описанного набором {OutputPin} (Method.return)}*.

Будем рассматривать методы ВМ *OEngine* как функции следующего вида:

- *behavior:* $X \times Y \times O \rightarrow Y \times O \times Z$ – вызов вычислительной команды *invoke(Context input [read only], Context context [read\update], Context return [update only], Object object [read\update], UltimateBehavior behavior)*;

− $operation_{parameter}: X \times Y \times O \rightarrow Y \times O \times Z$ – вызов метода *invoke(Context input [read only], Context context [read\update], Context return [update only], Object object [read\update], Object parameter, Operation operation)*;

− $new_{class}: Y \times O \rightarrow Y \times O \times Z$ – создание объекта заданного класса с сохранением его в рабочем контексте под заданным именем *createObject(Context context [read\update], Context return [update only], Object object [read\update], Class class)*;

− $constraint: X \times Y \times O \rightarrow \{0,1\}$ – проверка условия *verify(Context input [readonly], Context context [readonly], Object object, Constraint constraint)*.

Тогда функции переходов, вычислений и выходов определяются с их помощью следующим образом:

− функция выходов μ определяется как:

$$\mu(x, s, (y, o)) = z \Leftrightarrow [\exists\, CallBehaviorAction\ n, \exists y' \in Y, \exists o' \in O :$$
$$n = s \textbf{ and}$$
$$(y', o', z) = n.behavior(x, y, o)\ ]$$
$$\textbf{or } [\exists\, CallOperationAction\ n, \exists y' \in Y, \exists o' \in O :$$
$$n = s \textbf{ and}$$
$$(y', o', z) = n.operation_{n.parameter}(x, y, o)\ ]$$
$$\textbf{or } [\exists\, CreateObjectAction\ n, \exists y' \in Y, \exists o' \in O :$$
$$n = s \textbf{ and}$$
$$(y', o', z) = new_{n.class}(y, o)]$$
$$\textbf{or } [\exists\, DecisionNode\ n : n = s$$
$$\textbf{and } z = \varnothing\ ]$$
$$\textbf{or } [\exists\, InitialNode\ n : n = s$$
$$\textbf{and } z = \varnothing\ ];$$

− функция переходов λ определяется следующим образом:

$$\lambda(x, s, (y, o)) = s' \Leftrightarrow \exists\, ActivityEdge\ e :$$

e.source = s ***and***

[e.guard = null ***or*** *e.guard(x, y, o)]* ***and***

e.target = s';

- функция вычислений ν определяется следующим образом:

$$\nu(x, s, (y, o)) = (y', o') \Leftrightarrow [\exists\, CallBehaviorAction\ n, \exists\, z \in Z:$$

$$n = s \text{ and}$$

$$(y', o', z) = n.behavior(x, y, o)\ ]$$

$$\text{or } [\exists\, CallOperationAction\ n, \exists\, z \in Z:$$

$$n = s \text{ and}$$

$$(y', o', z) = n.operation_{n.parameter}(x, y, o)]$$

$$\text{or } [\exists\, CreateObjectAction\ n, \exists\, z \in Z:$$

$$n = s \text{ and}$$

$$(y', o', z) = new_{n.class}(y, o)]$$

$$\text{or } [\exists\, DecisionNode\ n: n = s$$

$$\text{and } (y', o') = (y, o)\ ]$$

$$\text{or } [\exists\, InitialNode\ n: n = s$$

$$\text{and } (y', o') = (y, o)\ ]$$

Таким образом, в данной главе были описаны три абстрактные программные модели для разных парадигм программирования, и было показано, как они реализуют императивную теоретико-множественную модель. Теперь, когда формализованы программные модели, можно провести их сравнительный анализ и, в частности, изучить возможность преобразования программ различных моделей, сохраняющие поведение.

3. ПРОЦЕДУРНОЕ ПРОГРАММИРОВАНИЕ И АВТОМАТНЫЙ ПОДХОД

Рассмотрим программы первых двух описанных моделей императивного программирования – автоматной и процедурной – и исследуем возможность преобразования программы из одной модели в программу в другой модели, реализующую то же поведение.

3.1. Преобразование «Процедура-Автомат»

В данной главе исследуется преобразование программ из процедурной модели в автоматную модель, обозначаемое как преобразование «Процедура-Автомат» и реализуемое алгоритмом *PATransformation*.

3.1.1. Алгоритм PATransformation

Опишем алгоритм *PATransformation*, преобразовывающий диаграмму деятельности в диаграмму состояний. Применив его к каждой процедуре исходной программы, получим преобразование «Процедура-Автомат».

При описании алгоритма будем использовать *незамкнутые* диаграммы, не имеющие начальных и заключительных вершин и содержащие произвольное количество входных или выходных висячих дуг. Любая такая диаграмма может быть замкнута, если добавить начальную вершину, соединив с ней все входы, и заключительные вершины на все выходящие висячие дуги (рис. 16).

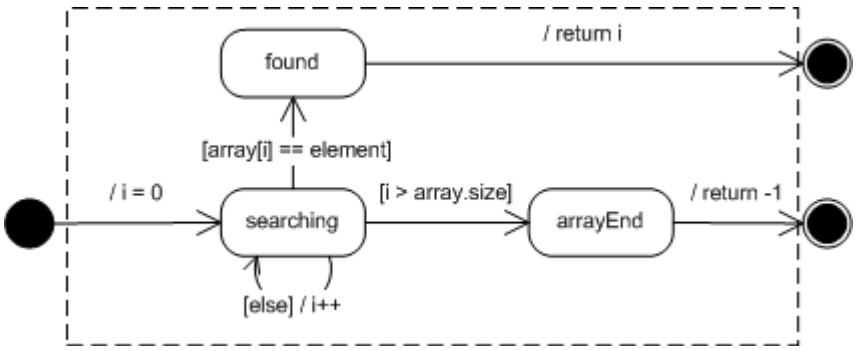

Рис 16. Замыкание диаграммы состояний (рамка не является частью нотации)

Алгоритм PATransformation:

Шаг 1. Построим гомеоморфную основу [24, 25] диаграммы деятельности.

Шаг 2. Построим реберный граф гомеоморфной основы.

Шаг 3. Преобразуем пометки.

В результате третьего шага алгоритма получим диаграмму состояний, эквивалентную по поведению исходной диаграмме деятельности.

Первые два шага алгоритма рассматривают входную диаграмму как граф с вершинами *ActivityNode*, которые соединены дугами *ActivityEdge*. Остальные элементы диаграммы считаются пометками на дугах (*constraint*) или вершинах (*behavior*). При построении гомеоморфной основы сделаем допустимым пометку вершин, полученных из вершин *CallBehaviorAction*, произвольным упорядоченным набором пометок *behavior*, после чего «склеим» все последовательные наборы таких вершин и добавим пустые (с пустым набором *behavior*) вершины между каждой парой соседних условных блоков *DecisionNode* (рис. 17).

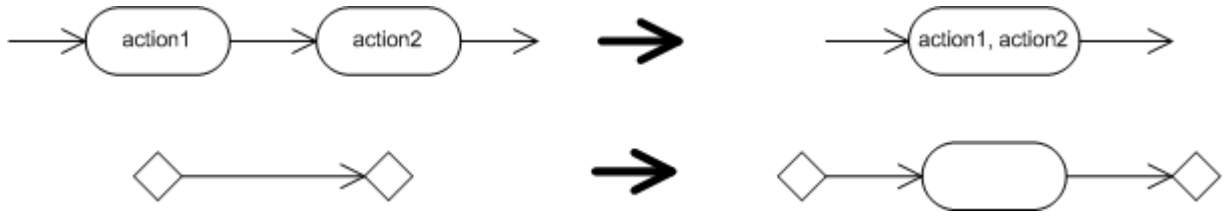

Рис 17. Построение гомеоморфной основы

Теперь, считая условные блоки *DecisionNode* гипердугами с произвольным числом входов и выходов, построим реберный граф для полученной гомеоморфной основы, сохраняя пометки на его элементах. При этом реберный граф будет определять состояния и переходы диаграммы состояний следующим образом: вершины реберного графа соответствуют состояниям *State*, дуги – переходам *Transition*.

В результате первых двух шагов алгоритма получим преобразование, сопоставляющее каждой условной вершине *DecisionNode* состояние *State*, а каждому набору последовательных действий – переход *Transition*.

Проиллюстрируем описанные действия алгоритма на примере программы, сортирующей массив (рис. 18).

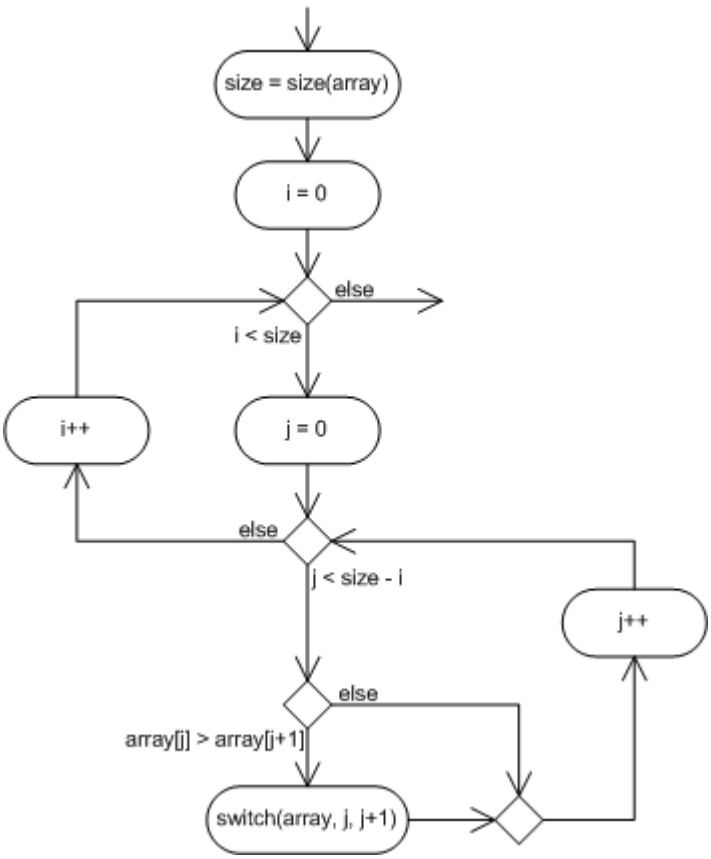

Рис 18. Незамкнутая процедурная программа сортировки массива: *sort(array)*

Гомеоморфная основа диаграммы процедуры *sort* представлена на рис. 19. Цветом выделены вершины, являющиеся объединением вершин или дополнением к исходному графу.

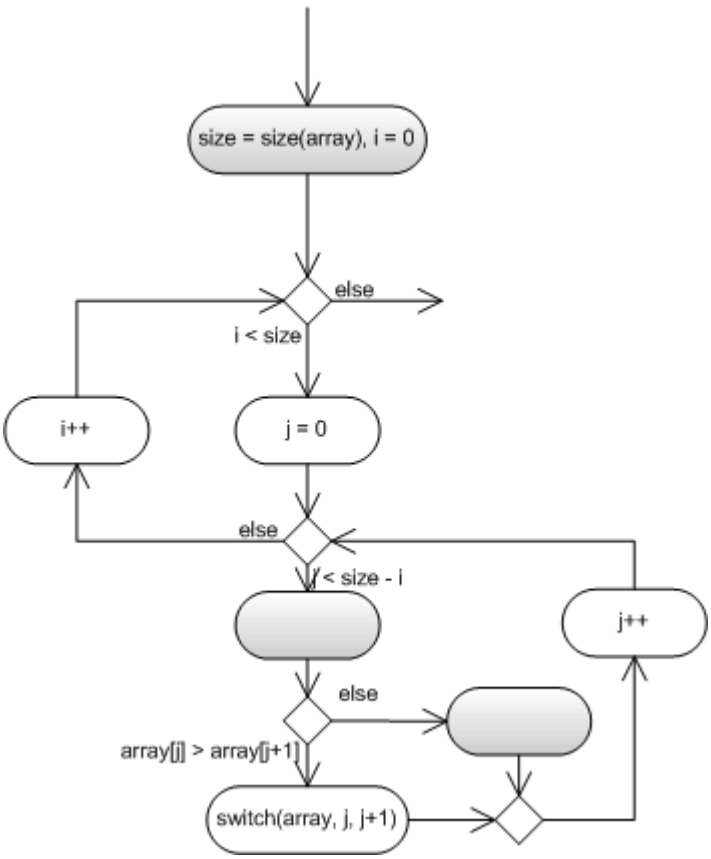

Рис 19. Гомеоморфная основа процедуры *sort(array)*

На рис. 20 показан реберный граф этой гомеоморфной основы.

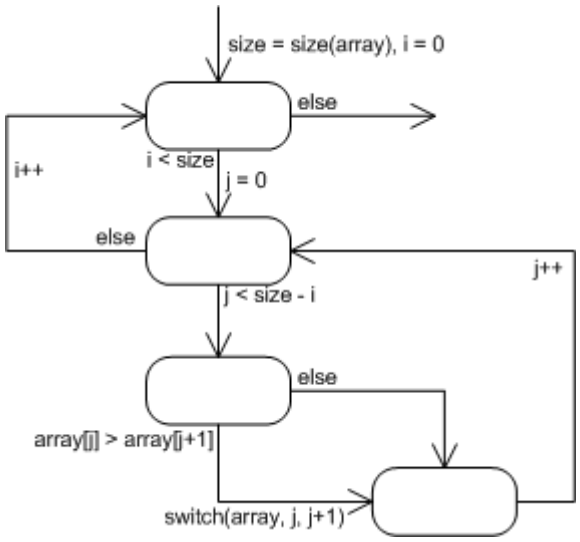

Рис 20. Реберный граф процедуры *sort(array)*

Третий шаг описываемого алгоритма достаточно тривиален. Как говорилось выше, при построении гомеоморфной основы и реберного графа были сохранены все пометки на преобразуемых элементах, в результате чего на полученной диаграмме переходы *Transition* помечены упорядоченным набором пометок типа *Behavior*, который и определяет элемент *Transition.effect*, а

выходы из состояния *State* помечены условиями типа *Constraint*, которые и определяют элемент *Transition.guard* (рис. 21). Такая простота обусловлена тем, что все рассматриваемые модели императивного программирования строятся на общем наборе элементарных команд, а их ВМ имеют аналогичные методы для их исполнения.

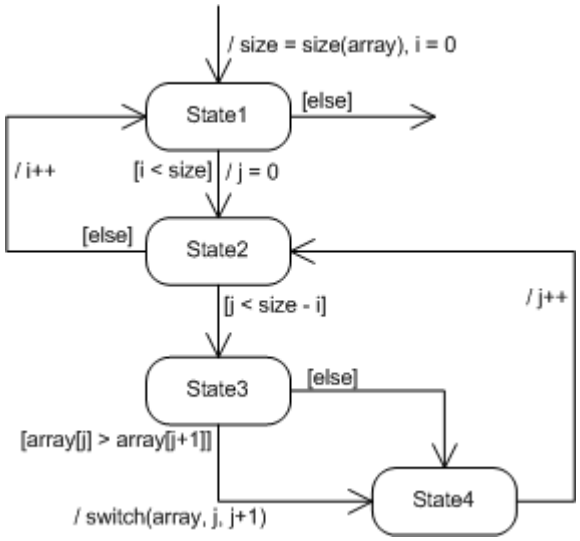

Рис 21. Незамкнутая диаграмма состояний автомата *sort(list)*

Замкнув полученную диаграмму состояний, получим результат преобразования.

3.1.2. Оценка алгоритма PATransformation

Пусть размер диаграммы деятельности определяется как $N + E$, где $N = \#\{ActivityNode\}$, $E = \#\{ActivityEdge\}$, $N_c = \#\{CallBehaviorNode\}$, $N_d = \#\{DecisionNode\}$, $E_d = \#\{ActivityEdge \mid (source.class = DecisionNode\}$, $E_i = \#\{ActivityEdge \mid (source.class = InitialNode\}$.

Тогда вычислительная сложность трех шагов алгоритма *PATransformation* измеряется как $O(E)+O(N+E)+O(N+E) = O(N+E)$. Емкостная сложность алгоритма также будет равна $O(N+E)$. Таким образом, алгоритм *PATransformation* задает *линейное* относительно размера исходной программы преобразование исходного поведения в диаграмму состояний.

Пусть размер диаграммы состояний выражается как $S + T + E$, где $S = \#\{Vertex\}$, $T = \#\{Transition\}$, $B = \#\{Behavior\ Transition.effect\}$.

Тогда размер полученной в результате преобразования диаграммы выражается как:

$S = N_d$;

$T = E_d + E_i$;

$B = N_c$.

Таким образом, размер полученной программы линейно соответствует размеру исходной.

3.2. Преобразование «Автомат-Процедура»

Рассмотрим теперь возможность обратного преобразования – получение процедурной программы по автоматной.

3.2.1. Алгоритм APTransformation

Для описанного выше алгоритма *PATransformation* можно построить обратный алгоритм, также основанный на графовых преобразованиях и имеющий тривиальный третий шаг, преобразующий пометки на графе. В результате получим преобразование автоматной программы в процедурную.

Алгоритм APTransformation:

Шаг 1. Построим реберный граф диаграммы состояний.

Шаг 2. Построим гомеоморфный граф диаграммы деятельности, удалив пустые и разделив множественные вершины *CallBehaviorAction*.

Шаг 3. Преобразуем пометки.

Заметим, что, применяя композицию прямого и обратного преобразований, получим диаграмму, полностью соответствующую исходной.

3.2.2. Оценка алгоритма APTransformation

Вычислительная сложность алгоритма *APTransformation* измеряется как *O(S+T+B)*, при такой же емкостной сложности *O(S+T+B)*. Таким образом, алгоритм *APTransformation* задает *линейное* относительно размера входной программы преобразование.

Размер полученной в результате преобразования диаграммы деятельности выражается следующим образом:

$N = S + B;$

$E = T + B.$

Таким образом, размер полученной программы линейно соответствует размеру исходной.

3.3. Доказательство сохранения поведения

Покажем, что преобразования алгоритмов *PATransformation* (обозначим его как ψ) и *APTransformation* (ψ^{-1}) сохраняют поведение.

Для начала заметим, что императивное поведение, реализованное автоматной программой, представляет функции μ, λ и ν с использованием композиции вызовов команд (поведений) *Transition.effect*. Очевидно, что такую композицию можно разделить на несколько императивных итераций, то есть заметить композицию команд на композицию μ, λ и ν. Тогда получим следующее *эквивалентное* императивное представление автоматной программы:

$P = (Y \times S, X, Z, \mu, \lambda, \nu, s_{нач}, S_{закл})$, где:

- $S := \{? \text{ extends } Vertex\} \cup \{Transition.effect\}$ – множество управляющих состояний императивного поведения;
- $s_{нач} := Pseudostate$ – начальное управляющее состояние совпадает с начальным состоянием автомата;
- $S_{закл} := \{FinalState\}$ – заключительные состояния;
- X, Y и Z задаются без изменений как показано в предыдущей главе;
- функция выходов μ определяется следующим образом:

$$\mu(x, s, y) = z \Leftrightarrow [\, \exists\, Vertex\ state : state = s \ \textbf{and}\ z = \varnothing\,]$$

$$\textbf{or}\ [\exists\, Transition\ t, \exists\, i \in N :$$

$$t.effect_i = s\ \textbf{and}$$

$(y', z) = t.effect_i(x, y)]$;

- функция переходов λ определяется следующим образом:

$$\lambda(x, s, y) = s' \Leftrightarrow [\exists\, Transition\ t:$$
$$t.source = s\ \textbf{and}$$
$$(t.guard = null\ \textbf{or}\ t.guard(x, y))\ \textbf{and}$$
$$(t.effect.size = 0\ \textbf{and}\ t.target = s'$$
$$\textbf{or}\ t.effect.size > 0\ \textbf{and}\ t.effect_0 = s')]$$
$$\textbf{or}\ [\exists\, Transition\ t, \exists\, i \in N:$$
$$t.effect_i = s\ \textbf{and}$$
$$(t.effect.size \leq i+1\ \textbf{and}\ t.target = s'$$
$$\textbf{or}\ t.effect.size > i+1\ \textbf{and}\ t.effect_{i+1} = s')];$$

- функция вычислений ν определяется следующим образом:

$$v(x, s, y) = y' \Leftrightarrow [\exists\, Vertex\ state: state = s\ \textbf{and}\ y' = y]$$
$$\textbf{or}\ [\exists\, Transition\ t, \exists\, i \in N:$$
$$t.effect_i = s\ \textbf{and}$$
$$(y', z) = t.effect_i(x, y)]$$

Для того, чтобы показать, что алгоритм сохраняет поведение, рассмотрим два типа элементарных незамкнутых диаграмм деятельности.

Тип 1: диаграмма, состоящая из единственной вершины и двух висячих дуг, обозначающаяся как *CallBehaviorAction[Behavior]* (рис. 22).

Тип 2: диаграмма, состоящая из условного блока, *k* входящих и *m* выходящих дуг, обозначающаяся как *DecisionNode[k, <m Constraint на выходах>]* (рис. 22).

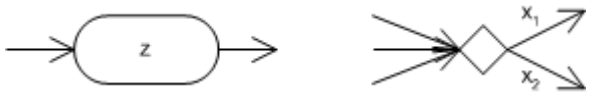

Рис 22. Элементарные фрагменты *CallBehaviorAction[z]* (слева) и *DecisionNode[3, x1, x2]* (справа)

Каждый такой фрагмент по построению алгоритма преобразуется в элементарный фрагмент диаграммы состояний:

effect[z]= ψ(CallBehaviorAction[z]) и (1)

State[3, x1, x2] = ψ(DecisionNode[3, x1, x3]). (2)

Полученные после преобразования элементарные фрагменты показаны на рис. 23.

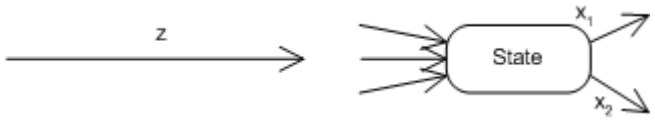

Рис 23. Элементарные фрагменты *effect[z]* (слева) и *State[3, x1, x2]* (справа)

Аналогичным образом определятся фрагменты, образующиеся при замыкании схемы:

Pseudostate = ψ(InitialNode) и (3)

FinalState = ψ(FinalNode). (4)

Пусть диаграмма деятельности *Activity* реализует некоторое поведение $f: X \to Y$, а диаграмма, полученная после работы алгоритма *PATransformation* – поведение $\psi(f) = f' : X \to Y$. Необходимо показать, что f эквивалентна f', что равносильно эквивалентности их императивных представлений.

Пусть f представлено в императивном виде как $P = (Y \times S, X, Z, \mu, \lambda, \nu, s_{нач}, S_{закл})$, а f' – как $P' = (Y \times S', X, Z, \mu', \lambda', \nu', s'_{нач}, S'_{закл})$. Из (1) и (2) следует, что обратимое преобразование ψ описывает отношение эквивалентности между множествами S и S', из (3) и (4) следует, что ψ описывает отношение эквивалентности между $s_{нач}$ и $s'_{нач}$, $S'_{закл}$ и $S'_{закл}$. Покажем теперь, что при таком отношении эквивалентности функции λ, μ и ν будут эквивалентны преобразованным функциям μ', λ' и ν'.

Например, покажем, что $\mu(x, s, y) = z$ влечет $\mu'(x, \psi(s), y) = z$. Первое равенство по определению μ означает, что выполняется одно из следующих условий:

[∃ CallBehaviorAction n : n = s ***and*** *(y', z) = n.behavior(x, y)],*

[∃ DecisionNode n : n = s ***and*** *z = ∅],*

[∃ InitialNode n : n = s ***and*** *z = ∅].*

Из этих условий следует по построению алгоритма (1–4), что выполняется одно из условий:

[∃ Transition t, ∃ i ∈ N: t.effect$_i$ = s ***and*** *(y', z) = t.effect$_i$(x, y)],*

[∃ Vertex state ***:*** *state = ψ(s)* ***and*** *z = ∅],*

что дает равенство $\mu'(x, \psi(s), y) = z$ по определению μ'.

Замечание 11. Как говорилось ранее, все описываемые модели обладают общим набором элементарных команд и эквивалентно реализуют их с помощью виртуальных машин. Отсюда получаем использованное выше утверждение, что *n.behavior(x, y)= t.effect$_i$(x, y)* (*PEngine.invoke(x, y, behavior) = AEngine.invoke(x, y, effect)* при условии, что *behavior = effect*).

Аналогично показывается обратное утверждение для преобразования ψ^{-1}, а также эквивалентность функций переходов λ и λ' и функций вычислений ν и ν':

$\lambda(x, s_1, y) = s_2 \Leftrightarrow \lambda'(x, \psi(s_1), y) = \psi(s_2)$,

$\nu(x, s, y_1) = y_2 \Leftrightarrow \lambda'(x, \psi(s), y_1) = y_2$.

Таким образом, между управляющими состояниями императивных поведений *P* и *P'* было построено взаимно-однозначное соответствие, после чего было показано, что функции переходов, вычислений и выходов в исходных состояниях и в эквивалентных им состояниях преобразованной программы, выдают одинаковый (эквивалентный) результат. Следовательно, поведение *f*, реализованное диаграммой деятельности, и поведение *f'*, реализованное полученной преобразованием «Процедура-Автомат» диаграммой состояний, эквивалентны.

3.4. Сравнение с существующими аналогами

Рассматриваемый в данной главе тип преобразований неоднократно упоминался в различных модификациях и с различной степенью формализации.

Например, в книге [26] предлагается алгоритм преобразования микропрограмм, задаваемых граф-схемами (аналог блок-схемы или диаграммы деятельности), в автоматы Мили. Описание предложенного в [26] алгоритма, также как и его сравнение с алгоритмом, описанным в настоящей работе, представлено в табл.1.

Таблица 1. Сравнение алгоритма *PATransformation* с существующим аналогом

Параметр сравнения	Алгоритм [26]	*PATransformation*
описание синтаксиса входной программы	словесное, основанное на стандарте блок-схем	диаграммой классов, основанное на стандарте *UML*
отличия в синтаксисе входной программы	нет вложенных вызовов, в операторных вершинах только элементарные микрокоманды; произвольное количество входов у операторных вершин; ровно два выхода у условной вершины	допустимы вложенные (и рекурсивные) вызовы; единственный вход у операторной вершины; произвольный набор выходов у условной вершины
описание семантики входной и выходной программ	словесное	абстрактная ВМ
описание синтаксиса выходной программы	словесное	диаграммой классов
синтаксис выходной программы	нет вложенных вызовов; единственное выходное воздействие на переходе	допустимы вложенные (и рекурсивные) вызовы; произвольное число выходных воздействий

		на переходе
описание алгоритма преобразования	теоретико-множественное	теория графов
основа алгоритма построения	сопоставляет состояние операционной вершине	сопоставляет состояние условной вершине
точные оценки сложности алгоритма	не предоставлены, при подсчете оказывается, что они будут ***экспоненциальными***	есть, ***линейные***
точные оценки размера полученной программы	не предоставлены	есть, линейные

Не умаляя ценности предложенного в [26] алгоритма, заметим, что он преследует другую цель, и, будучи ориентированным именно на микропрограммы, не позволяет исследовать вопрос преобразования императивных парадигм в контексте языков программирования высокого уровня, а именно эта задача является основной задачей данной работы.

Еще одна реализация алгоритма преобразования процедурной программы в автоматную описана в статьях [27, 28], основанных на работах [29, 30]. Отличия алгоритма *PATransformation* от описанного, например, в [28], показаны в табл. 2.

Таблица 2. Сравнение алгоритма PATransformation с аналогом [28]

Параметр сравнения	Алгоритм [28]	*PATransformation*
описание синтаксиса входной программы	грамматика	диаграмма классов
отличия в синтаксисе входной программы	имеет текстовую нотацию; описывает *только структурные* алгоритмы; вложенные	имеет графическую нотацию; описывает произвольные императивные

	вызовы описаны невнятно	алгоритмы; допускает вложенные вызовы
описание семантики входной и выходной программ	словесное	абстрактная ВМ
описание синтаксиса выходной программы	Грамматика	диаграмма классов
отличия в синтаксисе входной программы	вложенные вызовы реализованы нерационально, работа с памятью не описана	допускает вложенные вызовы
алгоритм: оценка трудоемкости или размера полученной программы	не предоставлены, а определить, будут отличаться, хотя останутся линейными.	есть
доказательство	словесное	теоретико-множественное

Алгоритм [28] преобразования процедурной программы в автоматную, в отличие от алгоритма [26], позволяет работать с процедурными программами на языках высокого уровня, однако его основной задачей является преобразование структурных алгоритмов в автоматные (задача, очень важная, например, при разработке визуализаторов). Абстрактный вопрос сложности преобразованных моделей и преобразования любой императивной программы не является целью работы [28].

Таким образом, предложенный в настоящей работе алгоритм, хоть и имеет ряд аналогов, является более полно и формально описанным, более рациональным и позволяет работать с более широким классом преобразуемых программ.

4. ООП И АВТОМАТНЫЙ ПОДХОД

Рассмотрим теперь преобразования поведения между объектно-ориентированной моделью и моделью автоматных служб. Последняя является модификацией автоматной модели, описанной в главе 2.

4.1. Модель автоматных служб

Представим программу в модели автоматных служб как набор автоматов, имеющих доступ к некоторым внешним (удаленным) данным. Пусть каждое автоматное поведение задается диаграммой состояний (*StateMachine*), как описано в главе 2. Таким образом, абстрактный синтаксис данной модели совпадает с синтаксисом автоматной модели. Опишем, как виртуальная машина и способ интерпретации программы в этой модели отличается от автоматной.

4.1.1. Виртуальная машина модели автоматных служб

Определим службу доступа к удаленным данным в виде некоторого контекста *RemoteContext* (рис. 24). В общем случае он может быть типизированным или нет, но во втором случае он должен хранить некоторый именной идентификатор name, позволяющий определить набор автоматов, которые могут работать с данным удаленным контекстом.

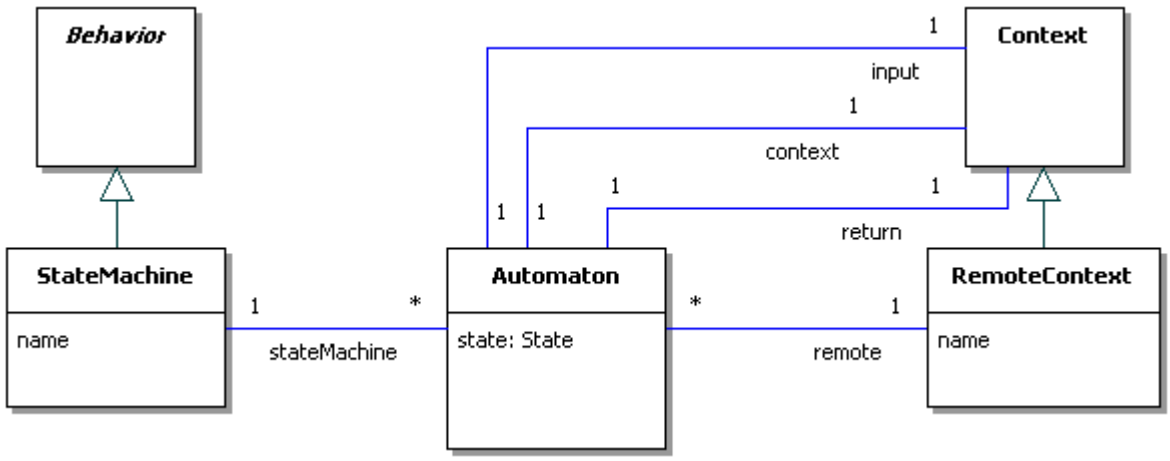

Рис 24. Программа во время выполнения в модели автоматных служб

Жизненным циклом удаленного контекста, также как и автомата, будет управлять виртуальная машина модели автоматных служб *ASEngine* (рис. 25).

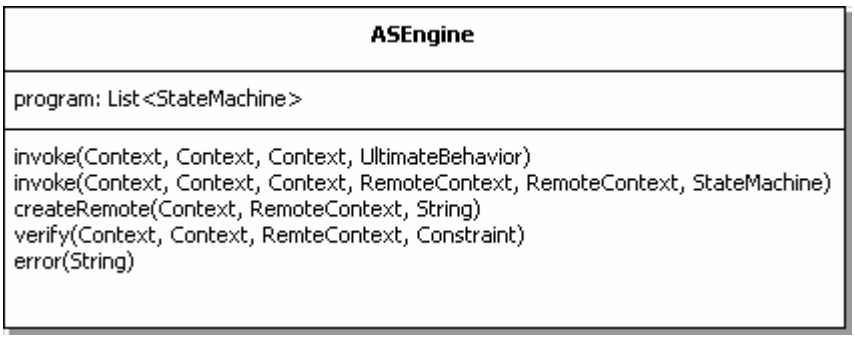

Рис 25. Виртуальная машина ASEngine

Данная виртуальная машина во многом схожа с виртуальной машиной *AEngine* и реализует следующие методы:

− вызов элементарной команды *invoke(Context input [readonly], Context context [read\update], Context return [updateonly], UltimateBehavior behavior)*;

− вызов автомата *stateMachine* на удаленном контексте *parameter invoke(Context input [readonly], Context context [read\update], Context return [updateonly], RemoteContext remote [read\update], RemoteContext parameter, StateMachine stateMachine)*;

− создание удаленного контекста с заданным именем с сохранением его в рабочем контексте автомата *createRemote(Context context [update], RemoteContext remote[update] String name)*;

− проверка условия *verify(Context input [readonly], Context context [readonly], RemoteContext remote, Constraint constraint)*.

Вспомним, что в качестве элементарной команды *UltimateBehavior* может выступать вычислительная команда любого вида. Пусть тогда методы ВМ (первые три метода) также могут выступать как вычислительные команды. Такое предположение и его описание в явном виде, следующее далее, будет удобно использовать при изучении возможности преобразование ОО модели в модель автоматных служб.

Таким образом, модель автоматных служб отличается от простой автоматной модели тем, что каждый экземпляр автомата имеет доступ к некоторому внешнему контексту данных, которые, наравне с данными

контекста автомата, могут быть использованы (на чтение и запись) при вызове элементарных команд *UltimateBehavior* или при вызове вложенных автоматных воздействий.

4.1.2. Императивность модели автоматных служб

Покажем, что программа, имеющая заданный синтаксис и интерпретируемая описанной абстрактной виртуальной машиной, является императивной. Рассматривая каждую службу программы как реализацию поведения, имеющего доступ (на чтение и запись) к некоторому внешнему контексту, можно представить каждое поведение как императивное, в котором вычислительное состояние состоит из пары компонентов – локальное (рабочий контекст) и удаленное (удаленный контекст). Таким образом, автоматом *StateMachine* в теоретико-множественной модели определяется императивное поведение $P = (S \times Y \times R, X, Z, \mu, \lambda, \nu, s_{нач}, S_{закл})$, где

- $S := \{? \text{ extends } Vertex\} \cup \{Transition.effect\}$ – набор управляющих состояний поведения;
- $s_{нач} := Pseudostate$ – начальное управляющее состояние;
- $S_{закл} := \{FinalState\}$ – набор заключительных команд;
- $Y := \{$*вычислит. состояния контекста произвольного вида (Automaton.context)*$\}$;
- $R = \{$*вычислит. состояния контекста RemoteObject*$\}$ – множество внешних вычислительных состояний;
- $X = \{$ *вычислит. состояния контекста, описанного {InputPin}}* - множество входных наборов;
- $Z = \{$ *вычислит. состояния контекста, описанного {OutputPin}}*.

Будем рассматривать методы ВМ *ASEngine* как функции следующего вида:

- $behavior: X \times Y \times R \rightarrow Y \times R \times Z$ – вызов элементарной команды *invoke(Context input [read only], Context context [read\update], Context return*

[update only], RemoteContext remote [read\update], UltimateBehavior behavior);

− $stateMachine_{parameter}$: $X \times Y \times R \rightarrow Y \times R \times Z$ – вызов автоматной службы *invoke(Context input [read only], Context context [read\update], Context return [updateonly], RemoteContex remote [read\update], RemoteContext parameter, StateMachine stateMachine)*;

− new_{name}:$Y \times R \rightarrow Y \times R$ – создание удаленного контекста с заданным именем *name* с сохранением ссылки на него в рабочем или удаленном контексте вызывающего автомата *createRemote(Context context [read\update], RemoteContext remote [read\update], String name)*;

− *constraint*:$X \times Y \times R \rightarrow \{0,1\}$ – проверка условия *verify(Context input [read only], Context context [read only], RemoteContext remote [read only], Constraint constraint)*.

Тогда функции переходов, вычислений и выходов определятся с их помощью следующим образом:

− функция выходов μ определяется так:

$$\mu(x, s, (y, r)) = z \Leftrightarrow [\, \exists\, Vertex\ state : state = s \ \textbf{and}\ z = \varnothing\,]$$
$$\textbf{or}\ [\exists\, Transition\ t, \exists\, i \in N : t.effect_i = s \ \textbf{and}$$
$$(t.effect_i = stateMachine_{parameter}$$
$$\textbf{and}\ \exists\, y' \in Y, \exists\, r' \in R :$$
$$(y', r', z) = t.effect_i(x, y, r))$$
$$\textbf{or}\ (t.effect_i = new_{name}\ \textbf{and}\ z = \varnothing)$$
$$\textbf{or}\ (t.effect_i = behavior\ \textbf{and}\ \exists\, y' \in Y, \exists\, r' \in R :$$
$$(y', r', z) = t.effect_i(x, y, r))\ ].$$

Здесь условия вида $t.effect_i = stateMachine_{parameter}$ означают, что команда $t.effect_i$, соответствует вычислительной команде *UltimateBehavior* вызова метода ВМ, реализующего поведение $stateMachine_{parameter}$. Аналогично для метода ВМ new_{name}. Условие $t.effect_i = behavior$ означает,

что вызывается любая другая вычислительная команда, не являющаяся вызовом ни одного из описанных методов ВМ;

- функция переходов λ определяется следующим образом:

$$\lambda(x, s, (y, r)) = s' \Leftrightarrow [\exists\, Transition\ t :$$
$$t.source = s \ \textbf{and}$$
$$(t.guard = null \ \textbf{or}\ t.guard(x, y)) \ \textbf{and}$$
$$(t.effect.size = 0 \ \textbf{and}\ t.target = s'$$
$$\textbf{or}\ t.effect.size > 0 \ \textbf{and}\ t.effect_0 = s')]$$
$$\textbf{or}\ [\exists\, Transition\ t, \exists\, i \in N :$$
$$t.effect_i = s \ \textbf{and}$$
$$(t.effect.size \leq i+1 \ \textbf{and}\ t.target = s'$$
$$\textbf{or}\ t.effect.size > i+1 \ \textbf{and}\ t.effect_{i+1} = s')];$$

- функция вычислений ν определяется следующим образом:

$$v(x, s, (y, r)) = (y', r') \Leftrightarrow [\exists\, Vertex\ state : state = s$$
$$\textbf{and}\ (y', r') = (y, r)\ ]$$
$$\textbf{or}\ [\exists\, Transition\ t, \exists\, i \in N : t.effect_i = s \ \textbf{and}$$
$$(t.effect_i = stateMachine_{parameter} \ \textbf{and}\ \exists\, z \in Z :$$
$$(y', r', z) = t.effect_i(x, y, r))$$
$$\textbf{or}\ (t.effect_i = new_{name}$$
$$\textbf{and}\ (y', r') = t.effect_i(y, r))$$
$$\textbf{or}\ (t.effect_i = behavior \ \textbf{and}\ \exists\, z \in Z :$$
$$(y', r', z) = = t.effect_i(x, y, r))\ ];$$

4.2. Преобразование «Метод-Автомат»

Рассмотрим преобразование поведения из объектно-ориентированной модели в модель автоматных служб, основанное на сопоставлении одного

автомата каждому методу ОО программы. Обозначим такое преобразования «Метод-Автомат».

4.2.1. Алгоритм MATransformation

Преобразование класса строится на преобразовании каждого метода в отдельности, в результате чего получается набор автоматов, пользующихся общим удаленным контекстом.

Замечание 12. Ассоциации между командой вызова вложенного поведения и описанием поведения (например, *CallBehaviorAction.behavior* типа *Activity*) при описании программ обычно реализуются через уникальные имена всех реализованных в программе поведений.

Так как в рамках ОО модели имя метода уникально для фиксированного класса, а для модели автоматных служб имя автомата уникально в пределах программы, оговоримся определять имена автоматам по следующему правилу:

<имя автомата> := <имя класса><имя метода>

Алгоритм полностью аналогичен алгоритму преобразования процедуры *PATransformation*, и отличается только реализацией третьего шага, а именно, преобразованием пометок.

Шаг 3. Преобразование пометок:

− заменим операцию создания объекта класса операцией (*ASEngine.new*) создания экземпляра удаленного контекста, сохранив в нем имя класса, то есть пометка вершины *CreateObjectAction* с именем *class* преобразуется в вызов операции *createRemote(<контексты автомата>, CreateObjectAction.parameter.name)*, что упрощенно можно изобразить так, как показано на рис. 26;

Рис 26. Преобразование создания объекта

− преобразуем вызов метода некоторого объекта в вызов автоматной службы (*ASEngine.stateMachine*) на удаленном контексте *RemoteContext*, соответствующем этому объекту (рис. 27);

Рис 27. Преобразование вызова метода

− при преобразовании элементарных команд *UltimateBehavior* обозначение доступа к полям класса (зависит от реализации элементарных команд) преобразуем в обозначение доступа к данным удаленного контекста автомата. Локальные переменные алгоритма, как и в алгоритме *PATransformation*, будут храниться в рабочем контексте автомата.

4.2.2. Доказательство сохранения поведения

Как и в доказательстве, приведенном в предыдущей главе, покажем, что императивные поведения, реализуемые исходной и преобразованной программами, эквивалентны.

Вспомнив, что первые два шага алгоритма совпадают с первыми шагами алгоритма *PATransformation*, получим отношение эквивалентности между множествами команд (всех команд поведения, начальных и заключительных команд) исходного и полученного поведений. Таким образом, осталось показать эквивалентность функций λ, μ и ν.

Для этого заметим, что в алгоритме *PATransformation* существовало три вида пометок на исходной диаграмме – пометка *CallBehaviorAction.behavior* типа *Activity*, пометка *CallBehaviorAction.behavior* типа *UltimateBehavior*, и пометка *Transition.guard* типа *Constraint*. Третий шаг алгоритма был тривиален, так как эти три вида пометок трансформировались в пометки того же типа (но преобразованного, в первом случае) – *StateMachine*, *UltimateBehavior* и *Constraint* соответственно, а виртуальные машины процедурной и автоматной модели реализуют одинаковые (семантически) методы для запуска команд, соответствующих этим пометкам.

В данном алгоритме преобразование пометок менее тривиально, а именно, если обозначить преобразование как ψ, получим:

$$\psi(CallBehaviorAction.behavior:UltimateBehavior) = \\ = Transition.effect: ASEngine.behavior \quad (5)$$

$$\psi(CallOperationAction.behavior:Operation) = \\ = Transition.effect: ASEngine.stateMachine \quad (6)$$

$$\psi(CreateObjectAction) = Transition.effect:ASEngine.new \quad (7)$$

$$\psi(ActivityEdge.constraint:Constraint) = Transition.guard:Constraint \quad (8)$$

Сравнивая определения функций μ, λ и ν в ОО модели и в модели автоматных служб и учитывая свойства алгоритма (5–8), получим, что преобразованные функции переходов, вычислений и выходов эквивалентны исходным.

4.2.3. Оценка алгоритма MATransformation

Как видно из описания алгоритма, его вычислительная и емкостная сложность полностью соответствуют оценкам алгоритма *PATransformation* – то есть они линейны относительно размера исходной программы.

Размер полученной программы также линейно соответствует размеру исходной, с теми же точными формулами, которые описаны при оценке алгоритма *PATransformation*.

4.3. Преобразование «Автомат-Метод»

Такого типа преобразования подробно не рассматриваются в рамках настоящей работы.

В работах [31–34] предлагается ряд методов реализации автоматов в рамках ООП, которые можно (однако не в формальном виде, предложенном данной работе) рассматривать как преобразование автоматного описания поведения в объектно-ориентированное описание. Такое преобразование обычно называют трансляцией автоматов и основывают на одном из следующих утверждений:

- автоматы, как методы класса,
- автоматы, как классы,
- автоматы, как классы с использованием базового класса.

Последний метод реализован, например, в рамках проекта *Unimod* [35] для автоматической трансляции автоматов в программу на языке *Java*.

Применяя любой из этих методов, получаем ОО программу, построенную по заданной системе автоматов. Все такие преобразования, как правило, имеют линейную сложность и в результате своей работы получают программу, размер которой линейно зависит от размера исходной.

ЗАКЛЮЧЕНИЕ

В работе рассмотрены три детерминированных модели наиболее популярных императивных парадигм программирования: процедурная, автоматная, объектно-ориентированная. Для каждой парадигмы приведена одна из возможных реализаций, задающая операционную семантику в форме абстрактной виртуальной машины. Для описания программ в каждой из парадигм использованы наиболее популярные в настоящее время средства графической нотации: диаграммы деятельности (блок-схемы), диаграммы состояний (графы переходов), расширенные диаграммы классов (диаграммы классов со встроенными диаграммами поведения). Для указанных диаграмм определены метрики, определяющие размер диаграммы. Введено понятие преобразования диаграммы (программы), сохраняющего поведение.

Показано, что для трех рассмотренных парадигм существуют преобразования, сохраняющие поведение, причем как трудоемкость преобразования, так и размер преобразованной программы ограничены сверху линейными функциями от размера исходной диаграммы.

Таким образом, показано, что три императивных парадигмы программирования линейно сводимы друг к другу.

В перспективе предполагается распространить полученные результаты на неимперативные парадигмы и недетерминированное поведение.

СПИСОК ЛИТЕРАТУРЫ

1. *Непейвода Н.Н.* Стили и методы программирования. М.: Интернет-Университет Информационных технологий, 2005. http://is.ifmo.ru/foundation/moving
2. *Pratt T.W., Zelkowitz M.V.* Programming Languages: Design and Implementation, 3rd ed. Englewood Cliffs, N.J.: Prentice Hall, 1996.
3. *Sebesta. R.W.* Concepts of Programming Languages, Mass.: Addison–Wesley Publishing Company, 1996.
4. *Непейвода Н.Н., Скопин И.Н.* Основания программирования. 2002.
5. *Шалыто А.А.* SWITCH-Технология. Алгоритмизация и программирование задач логического управления. СПб.: Наука, 1998. http://is.ifmo.ru/books/switch/1
6. *Любченко В.С.* Конечно-автоматная технология программирования. Труды международной научно-методической конференции «Телематика'2001». СПб.: СПбГИТМО(ТУ), 2001.
7. *Глушков В.М.* Синтез цифровых автоматов. М.: Физматгиз, 1962.
8. *Буч Г.* Объектно-ориентированный анализ и проектирование с примерами приложений на C++. СПб.: Невский диалект, 2001.
9. *Коуд П., Норт Д., Мейфилд М.* Объектные модели. Стратегии, шаблоны и приложения. М.: Лори, 1999.
10. *Айзерман М.А., Розонер Л.И., Смирнова И.М., Таль А.А.* Логика. Автоматы. Алгоритмы. М.: Физматгиз, 1963.
11. *Bock C.* UML 2 Activity and Action Models, in Journal of Object Technology, vol. 2, no. 4, July–August. 2003. http://www.jot.fm/issues/issue_2003_07/column3
12. *Bock C.* UML 2 Activity and Action Models Part 2: Actions, in Journal of Object Technology, vol. 2, no. 5. http://www.jot.fm/issues/issue_2003_09/column4

13. *Bock C.* UML 2 Activity and Action Models Part 3: Control Nodes, in Journal of Object Technology, vol. 2, no. 6. http://www.jot.fm/issues/issue_2003_11/column1/

14. *Bock C.* UML 2 Activity and Action Models Part 4: Object Nodes, in Journal of Object Technology, vol. 3, no. 1. http://www.jot.fm/issues/issue_2004_01/column3/

15. *Bock C.* UML 2 Activity and Action Models Part 5: Partitions, in Journal of Object Technology, vol. 3, no. 7. http://www.jot.fm/issues/issue_2004_07/column4/

16. *Bock C.* UML 2 Activity and Action Models, Part 6: Structured Activities, in Journal of Object Technology, vol. 4, no. 4, May–June 2005. http://www.jot.fm/issues/issue_2005_05/column4/

17. *Vitolins V., Kalnins A.* Semantics of UML 2.0 Activity Diagram for Business Modeling by Means of Virtual Machine. Proceedings Ninth IEEE International EDOC Enterprise Computing Conference, IEEE, 2005. http://arxiv.org/abs/cs/0509089/

18. *ГОСТ 19.002–80*. Схемы алгоритмов и программ. Правила выполнения. http://fmi.asf.ru/library/book/Gost/19-002-80-82.html

19. *ГОСТ 19.003–80*. Схемы алгоритмов и программ. Обозначения условные графические. http://fmi.asf.ru/library/book/Gost/19003-80-82.html

20. *ГОСТ 19.701–90*. Схемы алгоритмов, программ, данных и систем. Условные обозначения и правила выполнения. http://cert.obninsk.ru/gost/282/282.html

21. OMG Inc. UML 2.1.1 Superstructure Specification. http://www.omg.org/cgi-bin/doc?formal/07-02-03

22. *Буч Г., Рамбо Д., Джекобсон А.* Язык UML. Руководство пользователя. М.: ДМК, 2000.

23. *Фаулер М.* UML. Основы, 3-е издание. СПб.: Символ-Плюс, 2004.

24. *Вирт Н.* Алгоритмы и структуры данных. М.: Мир, 1989; С.-П.: Невский диалект, 2001.

25. *Кормен Т., Лайзерсон Ч., Ривест Р.* Алгоритмы. Построение и анализ. М.: МЦМНО, 2000.

26. *Баранов С.И.* Синтез микропрограммных автоматов (граф-схемы и автоматы). Л.: Энергия, 1979.

27. *Казаков М.А., Корнеев Г. А., Шалыто А.А.* Метод построения логики работы визуализатора алгоритмов на основе конечных автоматов. Телекоммуникации и информатизация образования. 2003, N6, с. 27-58.

28. *Корнеев Г.А.* Метод преобразования программ в систему взаимодействующих автоматов. «Труды II межвузовской конференции молодых ученых», СПб, 2005.

29. *Шалыто А.А., Туккель Н.И.* Преобразование итеративных алгоритмов в автоматные //Программирование. 2002. № 5. http://is.ifmo.ru/works/iter/

30. *Шалыто А.А., Туккель Н.И., Шамгунов Н.Н.* Реализация рекурсивных алгоритмов на основе автоматного подхода //Телекоммуникация и автоматизация образования. 2002. № 5. http://is.ifmo.ru/works/recurse/

31. *Шалыто А.А.*, *Наумов Л.А.* Методы объектно-ориентированной реализации реактивных агентов на основе конечных автоматов //Искусственный интеллект. 2004. №4. http://is.ifmo.ru/works/_aut_oop.pdf

32. *Корнеев Г.А., Шалыто А.А.* Реализация конечных автоматов с использованием объектно-ориентированного программирования // Труды X Всероссийской научно-методической конференции "Телематика-2003". 2003. Т.2.

33. *Шопырин Д.Г., Шалыто А.А.* Объектно-ориентированный подход к автоматному программированию. СПб.: СПбГУ ИТМО, 2003. http://is.ifmo.ru/projects/stool/

34. *Шалыто А.А., Туккель Н.И.* Танки и автоматы //BYTE/Россия. 2003. №2. http://is.ifmo.ru/works/tanks_new/

35. *Гуров В.С., Мазин М.А.* Инструментальное средство UniMod. http://UniMod.sourceforge.net/

36. *Дал У., Дейкстра Э., Хоор К.* Структурное программирование. М.: Мир, 1975.

37. *Bohm C., Jacopini G.* Flow Diagrams, Turing Machines And Languages With Only Two Formation Rules. Communications of the ACM. Volume 9, Number 5, May, 1966.

38. *Танненбаум Э.* Архитектура компьютера. СПб.: Питер, 2003.

39. *Ахо А., Сети Р., Ульман Д.* Компиляторы: принципы, технологии и инструменты. М.: Вильямс. 2001.

40. *Хантер Р.* Основные концепции компиляторов. М.: Вильямс. 2002.

41. *Гуров В.С., Мазин М.А.* Создание системы автоматического завершения ввода с использованием пакета UniMod /Вестник II межвузовской конференции молодых ученых. Т.1. СПб.: СПбГУ ИТМО. 2005.

42. *Шалыто А.А., Штучкин А.А.* Совместное использование теории построения компиляторов и SWITCH-технологии (на примере построения калькулятора). http://is.ifmo.ru/projects/calc/

43. *Potok T.E., Vouk M., Rindos A.* Productivity Analysis of Object-Oriented Software Developed in a Commercial Environment. Practice and Experience, Vol. 29, No. 10, 1999.