Санкт-Петербургский государственный политехнический университет
Физико-механический факультет
Кафедра прикладной математики

Диссертация допущена к защите
И.О. зав.кафедрой
________________ **В. Е. Клавдиев**

"__"___________________ 2008 г.

ДИССЕРТАЦИЯ
на соискание ученой степени МАГИСТРА

Тема: «*Определение языков программирования интерпретируемыми автоматами*»

Направление: 510200 – Прикладная математика и информатика
Магистерская программа: 510209 – Математическое и программное обеспечение вычислительных машин

Выполнила студентка группы 6057/2 У. Т. Тихонова

Руководитель, к. ф.- м. н. Ф. А. Новиков

Консультант по охране труда В. В. Монашков

Санкт-Петербург
2008

Реферат

В диссертации описан метод определения проблемно-ориентированных языков с помощью интерпретируемых автоматов. Этот метод позволяет формально определить проблемно-ориентированный язык в виде трех составляющих: абстрактного синтаксиса, конкретного синтаксиса и семантики. Программная реализация этого метода, машина автоматного программирования, осуществляет интерпретацию автоматов формального определения языка, и таким образом выполняет разбор и интерпретацию программ на данном языке. Метод описывается на примере языка предметной области *СЛОН*.

В работе рассмотрены существующие для описания проблемно-ориентированных языков методы и инструменты, проведен анализ диаграмм классов *UML* как средства задания языка, описаны автоматы, определяющие язык СЛОН, и методы их полуавтоматического построения.

Ключевые слова: проблемно-ориентированный язык, семантика, конкретный синтаксис, абстрактный синтаксис, метамодель, автомат, машина автоматного программирования.

Содержание

Введение ... 4
1. Обзор ... 7
1.1. Экономическая потребность решения задачи ... 7
1.2. Научный контекст ... 8
1.3. Технологический контекст ... 11
1.3.1. Языковый инструментарий Meta Programming System ... 12
1.3.2. Среда разработки MetaEdit+ Domain-Specific Modeling (DSM) ... 15
1.4. Технический контекст ... 17
1.4.1. Система ЭРА ... 17
1.4.2. Определение абстрактного синтаксиса с помощью диаграмм классов UML ... 19
1.5. Постановка задачи ... 19
2. Теоретическая часть ... 21
2.1. Основные положения ... 21
2.2. Определение абстрактного синтаксиса языка программирования с помощью диаграммы классов ... 22
2.2.1. Неформальный анализ диаграммы классов как метамодели проблемно-ориентированного языка ... 22
2.2.2. Метамодель языка СЛОН ... 25
2.3. Определение конкретного синтаксиса языка программирования с помощью распознающих автоматов ... 28
2.3.1. Использование парадигмы автоматного программирования ... 28
2.3.2. Система автоматов из метамодели: абстрактный конкретный синтаксис ... 30
2.3.3. Конкретный синтаксис и нотация языка СЛОН ... 32
2.4. Определение семантики языка программирования с помощью интерпретирующих автоматов 34
2.4.1. Семантика табличного сложения и умножения в языке СЛОН ... 35
2.4.2. Семантика табличных выражений языка СЛОН ... 39
2.4.3. Интерпретатор программы на языке СЛОН ... 41
3. Проектирование ... 45
3.1. Машина автоматного программирования ... 45
3.1.1. Структура автоматной программы ... 45
3.1.2. Графический синтаксис автоматной программы ... 47
3.1.3. Семантика автоматной программы ... 49
3.2. Интерфейс дерева программы ... 51
4. Реализация и применение машины автоматного программирования ... 54
4.1. Компонентная структура машины автоматного программирования ... 54
4.2. Транслятор автоматной программы ... 57
4.3. Реализация компонентов ... 59
4.4. Тестирование ... 62
Заключение ... 65
Литература ... 66
Приложение 1. Лог выполнения автоматной программы ... 68
Приложение 2. Лог выполнения автоматной программы в режиме раскрутки ... 71

Введение

Алгоритмическое решение любой задачи требует формализации ее в терминах некоторой предметной области, абстракции которой и допустимые операции над ними затем используются для построения непосредственно алгоритма решения. Это справедливо как для точных наук: таких как математика, физика, астрономия и других – так и для неточных наук (например, психология) и ненаучных задач. От выбора подходящей предметной области, от адекватности формализации задачи зависит сложность и корректность ее решения. В программировании решение задачи описывается в терминах языка программирования.

Языки программирования общего назначения универсальны и позволяют решить практически любую задачу. Но эта универсальность делает их довольно сложным инструментом, требующим высокой квалификации программиста и решения ряда общеизвестных проблем, возникающих при разработке прикладной программы. В отличие от языков программирования общего назначения, проблемно-ориентированные (или предметно-ориентированные) языки, известные также как языки предметной области (*Domain Specific Language, DSL*) создаются для решения некоторого ограниченного класса задач, специфичных для данной предметной области. Они позволяют перенести решение задачи на более высокий уровень абстракции по сравнению с уровнем языков программирования. Примерами таких языков являются *SQL, Maple, TEX.*

Стиль программирования, основанный на использовании языков предметной области, существует уже давно и известен сейчас как языкоориентированное программирование (*Language Oriented Programming, LOP*) [1]. Он обладает рядом приятных преимуществ, основанных на изложенной выше идее. Основным недостатком данного подхода является сложность создания нового языка и всего сопутствующего ему программного обеспечения. Одной из технологий, разработанных с целью устранить этот недостаток, является языковый инструментарий *Meta Programming System (MPS)* [2]. В его основе лежит постулат о том, что программа на языке предметной области – это любое точно определенное решение некоторой задачи, а не набор инструкций для компьютера. Общепринятое в практике программирования текстовое представление программы является лишь одним из множества представлений этого решения, и далеко не самым удобным...

Язык предметной области, как и язык программирования общего назначения, является средством выражения решения задачи. Он определяется синтаксисом и семантикой. Абстрактный синтаксис описывает класс допустимых языком программ (в

смысле решений некоторых задач). Конкретный синтаксис определяет представление для этого класса программ. Семантика придает программам содержание и смысл, определяет их поведение. Следовательно, описание языка программирования включает в себя определение всех этих трех составляющих. На практике обычно формально определяется только конкретный синтаксис языка (в виде его грамматики), а семантика и абстрактный синтаксис описываются неформально и скрываются в реализующий язык транслятор. Такой подход влечет немало неудобств как для использования языка, так и для его поддержки.

Одним из методов определения языков программирования, включающих в себя не только формальное описание конкретного синтаксиса языка, но и формальное определение его абстрактного синтаксиса и семантики, является *Венский метод* [3]. Для описания абстрактного синтаксиса в нем используется абстрактная грамматика, а для определения семантики языка – интерпретирующие автоматы.

На данный момент автоматы широко применяются как для описания и проектирования алгоритмов, так и для их реализации [0]. Программирование с использованием автоматов, или автоматное программирование, основано на работе интерпретирующей автоматы виртуальной машины. Согласно этого подхода, определение семантики языка с помощью автоматов является одновременно интерпретатором данного языка.

В данной работе рассматривается метод определения языков предметной области с помощью системы автоматов, интерпретируемых виртуальной машиной автоматного программирования. Структура языка, его абстрактный синтаксис, определяет представление конкретной программы, которое используют автоматы. Согласно наиболее популярной на данный момент объектно-ориентированной парадигмы программирования, структура языка описывается в виде диаграммы классов в нотации *UML*. Семантика языка задается системой автоматов, которые реализуют интерпретацию программы как экземпляра абстрактного синтаксиса. Конкретный синтаксис задается с помощью системы автоматов, реализующих разбор программы. При этом прототипом такой системы автоматов является структура языка. Автоматы интерпретируются виртуальной автоматной машиной, поэтому такая спецификация языка и есть программа, его реализующая.

Целью данной работы является исследование этого метода на примере описания спецификации языка *СЛОН* (*СЛежение и Обработка Наблюдений*) [4] системы *ЭРА* (*Эфемеридные Расчеты Астрономии*) – прикладной проблемно-ориентированной

системы программирования, предназначенной для решения разнообразных задач астрономии [5].

В первой главе рассматриваются существующие методы и инструменты определения проблемно-ориентированных языков, описываются основные особенности языка СЛОН и формулируется задача. Во второй главе приведен анализ выразительных средств рассматриваемого метода, разработан метод сведения фрагментов структуры языка к прототипам распознающих автоматов и определены абстрактный синтаксис, конкретный синтаксис и семантика для фрагмента языка *СЛОН*. В третье главе проектируется виртуальная машина автоматного программирования, реализующая метод определения проблемно-ориентированных языков с помощью интерпретируемых автоматов. В четвертой главе описаны реализация машины автоматного программирования и различные примеры ее использования.

1. Обзор

1.1. Экономическая потребность решения задачи

Создание программного обеспечения – это сложный и плохо управляемый процесс. Наиболее критичной частью этого процесса является непосредственно программирование приложения, так как программистам приходится работать с двумя предметными областями: предметной областью целевой задачи и используемым языком программирования. Программист осуществляет отображение одной предметной области в другую, поэтому от него требуется знание обоих предметных областей. Программирование в смысле такого отображения может быть упрощено с помощью языков предметной области (*DSL*). Языки предметной области позволяют решать целевую задачу в терминах этой задачи, а не в терминах вычислительной машины [2] и в идеале могут использоваться непосредственно специалистами в данной области. Такой подход не только облегчает разработку программ, но и как следствие повышает их качество.

Для того, чтобы достичь всех преимуществ языкоориентированного программирования, требуется его практическая реализация, позволяющая легко создавать и модифицировать языки предметной области и инструменты для работы с ними, а также предоставляющая возможность совмещать в разработке программы использование нескольких языков. Приложения, осуществляющие такую реализацию языкоориентированного программирования, называются языковым инструментарием [1]. Ключевым моментом при работе с языковым инструментарием является легкость его использования, так как главной задачей является практическая простота, а не теоретическая полнота.

Создание языкового инструментария – это достаточно широкая тема. Конкретная задача, решаемая в данной работе, – это исследование метода определения языков предметной области с помощью систем автоматов и рассмотрение возможных языковых инструментов, которые могут быть получены при этом автоматически. В дальнейшем на основе данного подхода может быть разработан языковый инструментарий.

Одним из преимуществ описываемого в данной работе метода является создание формальной спецификации не только синтаксиса разрабатываемого языка, но и его семантики. Такой подход облегчает использование нового языка, что, несомненно, важно для применения языкоориентированного программирования. При этом аналогично тому, как грамматика языка используется для генерации его синтаксического анализатора, формальное описание семантики может быть использовано как интерпретатор определяемого языка.

1.2. Научный контекст

Язык программирования как средство управления человека компьютером определяется тремя составляющими: синтаксисом (описывающим правила образования текстов программ), семантикой (определяющей связи между программами и их выполнением) и прагматикой (сопоставляющей создание программы ее назначению).

Прагматика языка, по сути, является методологией программирования, то есть описывает принципы, методы и приемы, позволяющие исходя из постановки задачи составить программу ее решения [6]. Многообразие решаемых задач, среди которых существуют задачи, неразрешимые стандартными средствами, не позволяет формализовать прагматику языка программирования.

С точки зрения синтаксиса язык программирования рассматривается как подмножество множества всех цепочек символов некоторого алфавита. Для определения этого, вообще говоря, бесконечного подмножества конечным путем широко применяется метод форм Бэкуса-Наура, или порождающих грамматик [7].

В книге [6] С. С. Лаврова описываются три основных подхода для формального определения семантики языков программирования:

1. *Операционный* подход: семантика языка описывается в виде интерпретатора его конструкций над некоторой (языково-ориентированной) абстрактной машиной.
2. *Аксиоматический* или *деривационный* подход: семантика описывается в виде системы аксиом, определенных для основных конструкций языка, и правил вывода, позволяющих получать из этих аксиом свойства программ, в том числе их выполнение.
3. *Денотационный* подход: семантика описывается с помощью модели алгоритма (интерпретации программ), выраженного в терминах некоторой совокупности множеств, отношений и отображений, которые описывают результаты исполнения программы в целом и ее отдельных частей.

Упомянутый выше *Венский метод* использует операционный подход, определяя семантику языка программирования интерпретирующими автоматами. Этот метод подробно описан в книге [3] А. Оллонгрена.

1.2.1. Венский метод (*Vienna Development Method*)

Венский метод формального описания языков программирования был разработан в 1968 году в Венской лаборатории корпорации *IBM* и был впервые применен для определения языка *PL/1*. Основной причиной разработки этого метода были недостатки

существовавших тогда описаний языков программирования и отсутствие критерия, позволявшего определить правильность реализации языка.

В книге [3] А. Оллонгрен рассматривает *Венский метод* исходя из классического определения формальных языков с помощью порождающих грамматик. Формальные языки и автоматы, решающие для них задачи распознавания и разбора, выражают только синтаксические аспекты определения языков программирования. Основанный на них синтаксический анализ программы позволяет получить некоторую структуру (известную как синтаксический граф или дерево вывода), которую затем можно интерпретировать. Однако дерево вывода содержит информацию, лишнюю на этапе интерпретации (например, различные ограничители: «;», «,», «begin», «end», «:=»), а также не полностью проявляет основную структуру программы (например, левая и правая части оператора присваивания не выражены явно, объявление переменных не отделено от списка операторов и т.д.). В *Венском методе* предлагается преобразовать программы в более удобные для интерпретации структуры с помощью обобщения синтаксических графов.

Любая система порождающих правил контекстно-свободной грамматики определяет класс объектов, имеющих те же структурные свойства, что и синтаксические графы (или деревья), порожденные этой грамматикой. Такие объекты называются абстрактными программами, а система предикатов, их определяющая, – абстрактной грамматикой. Таким образом, может быть построено преобразование из контекстно-свободной грамматики в абстрактную грамматику, описывающую структурные свойства программ, необходимые для семантического анализа и интерпретации.

Как абстрактные программы, так и состояния интерпретирующих автоматов определяются с помощью объектов и селекторов, представленных в виде размеченных (атрибутированных) деревьев. Эти понятия А. Оллонгрен вводит в описанной им теории структур данных, позволяющей ему формально описывать все преобразования и связи.

На рисунке 1.2.1.1. приведен пример синтаксического графа (слева) и соответствующего ему фрагмента абстрактной программы (справа) для выражения ((a∗b)/2). Здесь s1, s2, s3, s4, s5, s-rd1, s-rd2, s-op – селекторы.

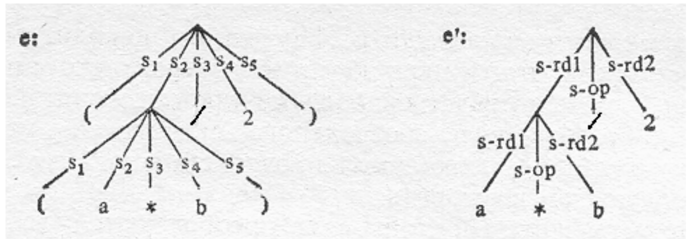


Рис. 1.2.1.1

Синтаксический граф (слева) и абстрактное представление (справа) для выражения ((a∗b)/2) [3]

Задача определения семантики языка решается путем придания смысла абстрактным программам. При этом абстрактная программа может быть получена как из конкретной программы (с помощью разбора и преобразования языка), так и с помощью порождающих правил абстрактной грамматики. Таким образом, описание семантики языка использует только абстрактный синтаксис и никак не зависит от конкретного синтаксиса языка.

Определяется класс автоматов, способных интерпретировать абстрактные программы (класс интерпретирующих автоматов, или интерпретаторов). Абстрактная грамматика вместе с интерпретирующим автоматом называется системой программирования.

В отличие от состояний классических конечных автоматов, состояния интерпретирующих автоматов, используемых в *Венском методе*, являются структурированными: состояние содержит компоненту управления, определяющую множество значений функции перехода, и компоненту хранилища, предназначенную для выполнения роли памяти. Непосредственная интерпретация языка осуществляется с помощью интерпретирующих функций, или схем команд, изменяющих обе компоненты состояний. Набор таких схем команд может рассматриваться как метапрограмма для интерпретирующего автомата.

Однако определенные таким образом интерпретаторы нельзя рассматривать как модели трансляторов языка (так как компонента управления в них – это структура данных состояния, изменяющаяся при выполнении). Чтобы получить формальное определение семантики, на основе которого может быть разработан реальный транслятор, задается система конечных автоматов, эквивалентная построенному ранее интерпретатору. Эта система называется абстрактным процессором. В нем управление отделено от элементов, использующихся для хранения данных.

Абстрактный процессор состоит из системы автоматов, обладающей следующими свойствами:

- каждый автомат имеет конечное число состояний, одно из которых является начальным и одно – заключительным. Состояния неструктурированы;
- один автомат называется начальным: его начальное и заключительное состояния являются соответственно начальным и заключительным состоянием системы;
- каждый автомат читает фиксированную абстрактную программу *p*, а также может использовать несколько элементов памяти для чтения и записи и работать с магазином;

- переходы из состояния в состояние детерминированы. При этом переходы могут быть безусловными (когда для данного состояния существует только один преемник), условными и вызовами подпрограмм (если преемник является начальным состоянием автомата системы).

На рисунке 1.2.1.2. приведен фрагмент абстрактного процессора языка *PL/1* из книги [3] – автомат интерпретации выражения. Здесь жирным шрифтом выделены названия автоматов (int-expr, int-bin-op, int-un-op). RS – магазин результатов. ǽ◦p – указатель, примененный к абстрактной программе p, определяет интерпретируемое выражение.

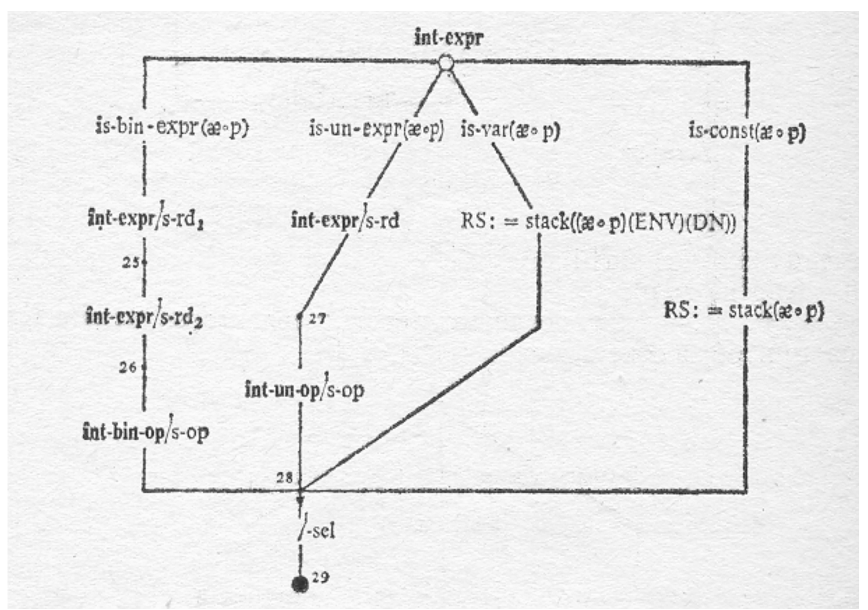


Рис. 1.2.1.2.

Автомат интерпретации выражения [3]

Таким образом, *Венский метод* вводит важное понятие абстрактной программы и отделяет описание семантики языка от его конкретного синтаксиса.

На данный момент *Венский метод* реализован языком формальной спецификации *VDM-SL* (*VDM Specification Language*). Существует ряд инструментов, постулирующих разработку, анализ, верификацию и кодогенерацию как для языков программирования, так и для программ, определенных с помощью *VDM-SL* [8].

1.3. Технологический контекст

Языковый инструментарий, появление которого вызвало новую волну интереса к языкоориентированному программированию, – это система *Meta Programming System* (*MPS*) компании *JetBrains* [2, 9, 10]. Схожую с языкоориентированным программированием концепцию моделирования в предметной области реализует *MetaEdit+ Domain-Specific Modeling Environment* – разработка компании *MetaCase* [11, 12].

Существуют и другие инструменты для создания языков предметной области (в частности для определения моделей и их различных представлений), например, *Generative Modeling Tools project* (*GMT*) [13], *Intentional Software* [14], *Microsoft Software Factories* [15]. Однако наиболее показательными с точки зрения используемых методов определения проблемно-ориентированных языков являются первые два инструмента.

1.3.1. Языковый инструментарий *Meta Programming System*

В статье [2] С. Дмитриев описывает свою реализацию языкоориентированного программирования (*MPS*), рассматривая программу как любое точно определенное решение некоторой задачи (а не набор инструкций для компьютера). На практике это означает, что программа – это некая абстрактная структура, которая имеет набор представлений (для хранения, отображения и выполнения) и которая редактируется непосредственно с помощью редактора (рис. 1.3.1.1).

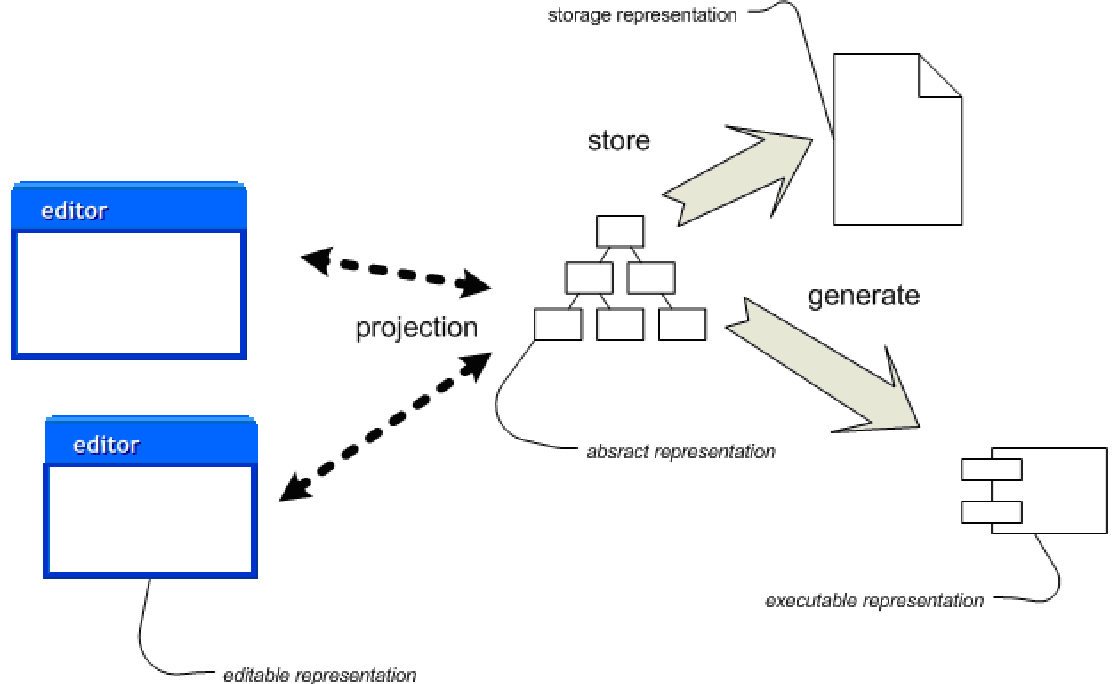


Рис. 1.3.1.1.

Манипулирование различными представлениями программы с помощью языкового инструментария [1]

Непосредственно программой является абстрактная структура. Для ее редактирования могут быть использованы различные представления: в виде текста, таблиц, диаграмм, формул и многие другие. При этом различные представления одной программы могут отражать ее различные аспекты. Для хранения программы используется отдельное представление. Генерация выполняемого файла из абстрактной структуры может легко расширяться для различных платформ и уровней абстракции.

Язык, позволяющий создать такую программу, определяется тремя составляющими:

- структурой (абстрактным синтаксисом);
- редактором (использующим конкретный синтаксис для отображения и редактирования);
- семантикой (позволяющей получить выполняемое представление программы).

Применяя концепцию языкоориентированного программирования, *MPS* позволяет создавать новые языки с помощью набора языков-конструкторов языков: язык описания структуры (*Structure Language*), язык описания редактора (*Editor Language*) и язык описания преобразования (*Transformation Language*).

Создание структуры нового языка – это перечисление всех типов (concepts) этого языка, которые обозначают используемые в нем понятия. Каждый тип определяется своими именем, свойствами и отношениями с другими типами (рис. 1.3.1.2). Соответственно, написание программы включает в себя создание экземпляров существующих типов и их связывание друг с другом.

```
BaseMethodDeclaration

Concept Declaration BaseMethodDeclaration

Extends: BaseSemanticType

Is Root: false

References:
returnType aggregation 1    1 target:Type                 specializes:<none>
parameter  aggregation 0..n 1 target:ParameterDeclaration specializes:<none>
body       aggregation 1    1 target:StatementList        specializes:<none>

Properties:
name       : _Identifier_String
visibility : Visibility
```

Рис. 1.3.1.2.

Определение типа «Method» с помощью Structure Language [2]

Редактор в *MPS* задается как набор клеток (cells), взаимное расположение которых, их структурированность (клетки могут состоять из клеток), способ задания (предопределенные ключевые слова или определенные пользователем идентификаторы), содержимое (текст, математические символы, векторная графика и т.д.), свойства (цвет) – все это используется для редактирования программы. Язык описания редактора позволяет задать клетки редактора (и их свойства) для каждого типа языка (рис. 1.3.1.3). Более того, можно добавить в редактор такую функциональность, как автозаполнение, рефакторинг и другие возможности современных *IDE*.

Рис. 1.3.1.3.

Определение редактора для типа «Method» с помощью Editor Language [2]

Для определения механизма выполнения программ на новом языке в *MPS* используется следующий метод. С помощью *Structure Language* определяется целевой язык, который может быть непосредственно транслирован в целевой формат программы (язык программирования, байт-код или машинный код). В этот целевой язык из исходного определяемого языка строится преобразование с помощью *Transformation Language*. Для описания этого преобразования используются вспомогательные механизмы:

- итеративный подход предполагает трансформацию каждого типа исходного языка в совокупность типов целевого языка;
- шаблоны позволяют генерировать целевой код из структуры исходного языка с помощью макросов (рис. 1.3.1.4);
- применение образцов проектирования для исходного языка помогает построить для него преобразование с помощью существующих «образцов преобразований».

Каждый из этих методов реализован в *MPS* с помощью соответствующего языка предметной области (*DSL*).

```
_PropertyDeclaration_Integer

Template Declaration _PropertyDeclaration_Integer

Parameters:
<< ... >>

Content node:
class _context_class_ extends SemanticNode {
    << fields >>
    <TF static String ${ _property_constant_ } = " ${ _property_value_ } " ; TF>
    << constructors >>
    <TF int ${ _getter_ } (     ) {                                          TF>
        return super. getIntegerProperty ( _context_class_ . _property_constant_ ) ;
    }
    <TF void ${ _setter_ } ( int value ) {                                    TF>
        super. setIntegerProperty ( _context_class_ . _property_constant_ , value ) ;
    }
    << static methods >>
}
```

Рис. 1.3.1.4.

Шаблон генерации кода на языке Java для типа «Property» [2]

Кроме того, в *MPS* существуют языки предметной области, которые могут пригодиться при разработке нового языка:

- базовый язык (*Base Language*) поддерживает стандартные средства языков общего назначения (арифметику, условные переходы, циклы и т.д.);
- язык коллекций (*Collection Languages*)
- язык описания интерфейса пользователя (*User Interface Language*)

Таким образом, *MPS* реализует инструмент для языкоориентированного программирования с помощью языкоориентированного программирования – применен метод раскрутки (bootstrapping).

1.3.2. Среда разработки *MetaEdit+ Domain-Specific Modeling (DSM)*

В книге [12] рассматривается концепция моделирования в предметной области (*Domain-Specific Modeling, DSM*). Она основывается на том же принципе, что и языкоориентированное программирование: языку моделирования общего назначения *UML* противопоставляется языки моделирования в предметной области. При этом такие языки моделирования в силу своей ограниченности (предметной областью) позволяют генерировать код с помощью моделей. Таким образом, по сути *DSM* поддерживает создание графических языков предметной области, которые могут не только выполняться, но и использоваться для обмена информацией, документооборота, создания статических структур (баз данных) – всего, что можно извлечь из моделирования. Авторы подробно освещают все преимущества и возможности, которые можно получить, используя данный подход, приводят статистические данные результатов его внедрения, исследуют основные принципы создания *DSL/DSM*.

MetaEdit+ DSM Environment реализует концепцию *DSM* с помощью двух инструментов: *MetaEdit+ Workbench* позволяет создавать языки моделирования предметной области, а *MetaEdit+ Modeler* поддерживает редактирование моделей в созданных языках.

Для определения языка моделирования в *MetaEdit+ Workbench* нужно определить понятия (concepts), накладываемые на них правила и ограничения, задать нотацию и описать генератор (рис. 1.3.2.1).

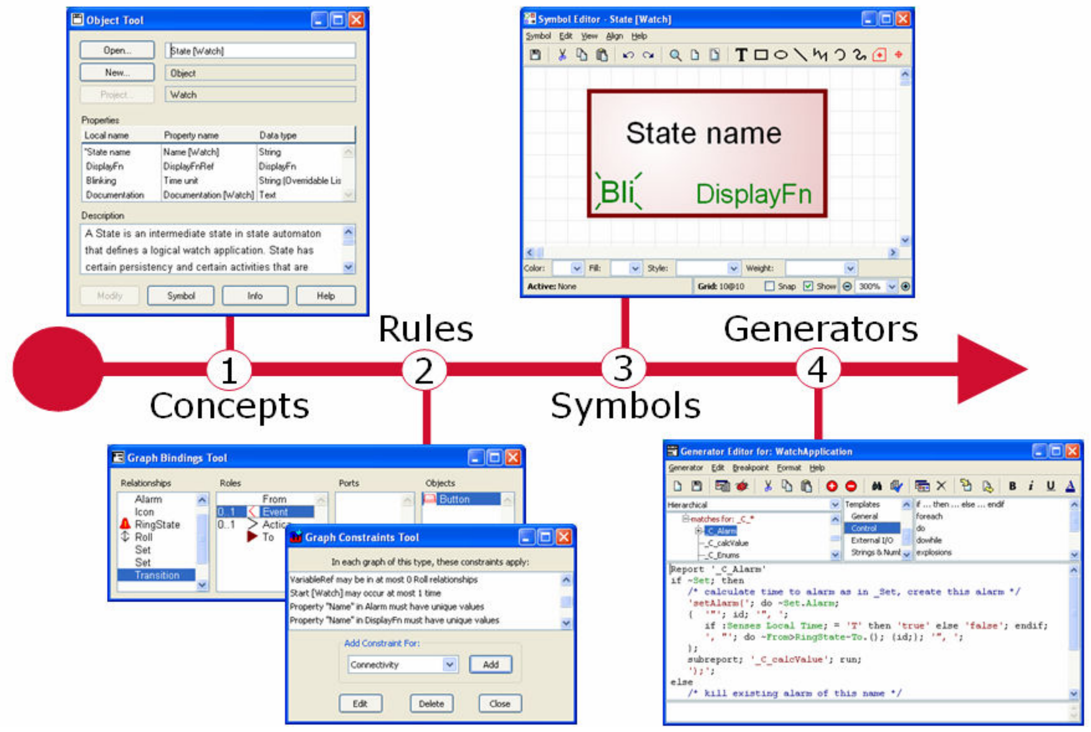


Рис.1.3.2.1.

Определение языка моделирования с помощью MetaEdit+ Workbench [11]

На первых двух шагах по сути создается абстрактный синтаксис нового языка. Основные понятия (типы) определяемого языка задаются с помощью инструмента метамоделирования в терминах так называемой *GOPPRR*-модели (*Graph-Object-Property-Port-Role-Relationship*). Таким образом, в данном контексте абстрактный синтаксис языка, или метамодель языка – это графоподобная структура, состоящая из объектов, обладающими некоторыми свойствами и связанными друг с другом ориентированными отношениями (при этом эти связи могут проходить через определенные порты). Накладываемые на метамодель ограничения и правила определяют свойства связей между объектами (кратность, порты, ориентацию), взаимные зависимости, число экземпляров и так далее. На третьем шаге для всех элементов метамодели языка рисуется нотация.

Моделироваться могут как статичные структуры, так и поведение. Для декларативного (информационно-логического) языка моделирования семантика статична и задается его метамоделью. Для выполняемого языка моделирования семантика

определяется поведением вычислительной модели. И в том и другом случае, в *MetaEdit+ Workbench* семантика использования разрабатываемого языка моделирования задается с помощью конкретного представления моделей, описываемых на нем.

С помощью редактора генераторов (*Generator Editor*) можно задать любое представление для модели как экземпляра метамодели языка: в виде документа *XML*, программы на известном языке программирования и т.д. Генераторы создаются с помощью предопределенных шаблонов, обращений к интерфейсу метамодели и скриптового языка описания генераторов *MERL* (рис. 1.3.2.2).

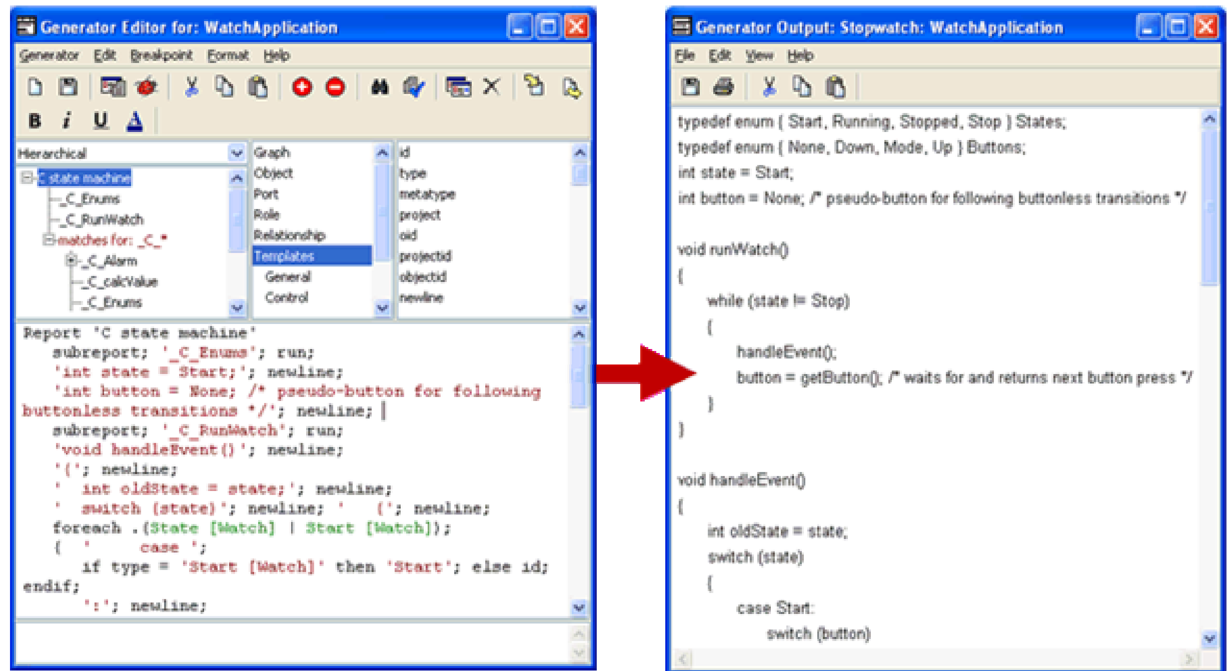


Рис. 1.3.2.2.

Редактор генераторов, язык MERL и полученный в результате генерации код программы [11]

Созданный язык затем используется для создания и редактирования моделей в *MetaEdit+ Modeler.* Таким образом, в основе определяемого языка лежит метаметамодель *GOPPRR* (модель для создания метамодели языка моделирования), а операционная семантика задается в генераторе.

1.4. Технический контекст

1.4.1. Система *ЭРА*

Рассматриваемый в данной работе метод определения проблемно-ориентированных языков исследуется на примере языка *СЛОН* (*СЛежение и Обработка Наблюдений*) [4] – входного языка системы *ЭРА* (*Эфемеридные Расчеты Астрономии*) [5]. *ЭРА* – это прикладная проблемно-ориентированная система программирования, предназначенная для решения разнообразных задач эфемеридной астрономии. Она

успешно используется уже более 20 лет. Система *ЭРА* базируется на табличном подходе к обработке данных и возможности настраивать предметную область с помощью описания конфигурации.

Табличный подход к обработке данных, или таблично ориентированное программирование [16] рассматривает таблицу с одной стороны как массив данных, с другой стороны как программу, определяющую последовательность присваивания значений переменным – величинам предметной области. Как массив данных таблица может храниться во внешней памяти (архиве). Как программа таблица – это итератор, перебирающий кортежи таблицы, чтобы выполнять с ними некоторые действия. При этом таблицами можно манипулировать с помощью табличных операций: сложения, умножения, итерации, проекции, выборки.

Описание конфигурации – это база данных, в которой собрано все, что может быть использовано при работе с системой *ЭРА*. Она содержит описание типов, переменных и действий, ссылаясь на конкретные модули функционального наполнения, которые по сути определяют семантику предметной области (рис. 1.4.1.1).

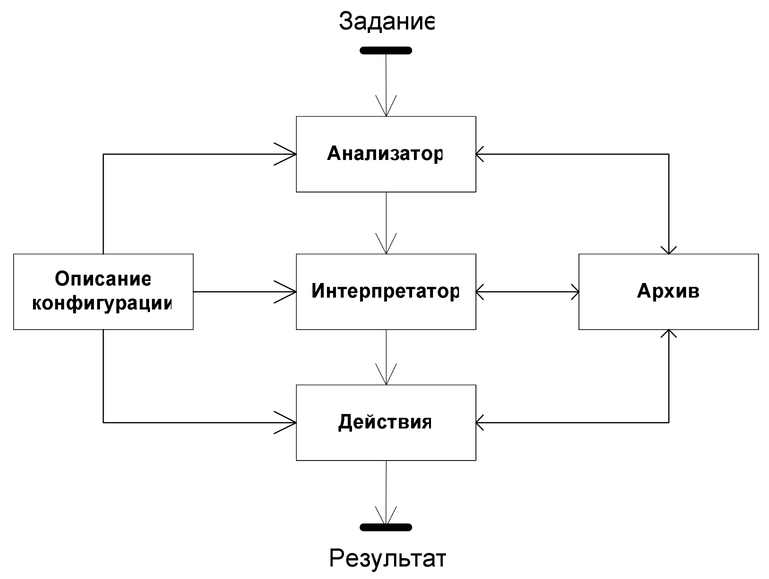


Рис. 1.4.1.1.
Структура системы ЭРА [5]

Проблемно-ориентированный язык *СЛОН* реализует таблично ориентированное программирование. Он успешно используется в настоящее время и является хорошим примером для исследования предлагаемого метода определения языков предметной области.

1.4.2. Определение абстрактного синтаксиса с помощью диаграмм классов *UML*

Рассматриваемый в данной работе метод использует для описания абстрактного синтаксиса языка диаграммы классов *UML*.

Определение абстрактного синтаксиса с помощью диаграмм классов называется метамоделированием. Эта техника активно используется организацией *Object Management Group (OMG)* при разработке объектно-ориентированных технологий и стандартов. Одним из примеров таких разработок является унифицированный язык моделирования *UML*. В основу его описания положен метод раскрутки: основные конструкции *UML* формально определены с помощью диаграмм классов *UML* [17]. При этом метамодель *UML* является экземпляром метамодели *Meta Object Facility (MOF)* – основного стандарта метамоделирования, который используется в технологии *Model Driven Architecture* (*MDA*).

Таким образом, определение абстрактного синтаксиса с помощью диаграмм классов позволяет использовать все концепции объектно-ориентированного моделирования, создавать гибкие и переносимые метамодели.

Для описания абстрактного синтаксиса с помощью диаграмм классов в предлагаемом методе определения языков используются следующие конструкции *UML*:

- классы – для представления понятий определяемого языка;
- атрибуты классов – для представления свойств понятий;
- обобщение – для классификации понятий;
- композиция и (реже) ассоциация – для представления отношений между понятиями.

1.5. Постановка задачи

Предлагаемый в данной работе метод определения языков программирования интерпретируемыми автоматами исследуется на примере описания языка *СЛОН*. Его реализация включает в себя выполнение следующих задач.

1. Определить абстрактный синтаксис (метамодель) языка *СЛОН* в виде диаграммы классов *UML* и в том числе:
    - провести анализ класса языков предметной области, для которых возможно описание с помощью метамодели в виде диаграммы классов;
    - провести анализ подмножества диаграмм классов, которые могут служить метамоделью языков предметной области;
2. Определить семантику языка *СЛОН* в виде системы интерпретирующих автоматов, использующих интерфейс метамодели и в том числе:
    - проанализировать потребовавшиеся от автоматов выразительные средства;

3. Свести метамодель языка к системе автоматов, анализирующих программу на языке *СЛОН*, и задать с помощью этих автоматов конкретный синтаксис:

- определить методы трансформации фрагментов метамодели языка во фрагменты распознающих автоматов;
- провести анализ конкретных синтаксисов, которые могут быть заданы на распознающих автоматах;

4. Разработать виртуальную машину автоматного программирования, интерпретирующую определенные ранее автоматы. На основе анализа определенных для языка СЛОН автоматов:

- определить модель автоматной программы;
- разработать алгоритм интерпретации автоматных программ;
- спроектировать и реализовать интерфейсы метамодели языка *СЛОН* и автоматной машины.

Таким образом, рассматриваемый проблемно-ориентированный язык *СЛОН* определяется в виде трех составляющих: абстрактного синтаксиса, семантики и конкретного синтаксиса. Виртуальная машина автоматного программирования позволяет рассматривать это определение языка *СЛОН* как реализующую его программу. При этом языковый процессор не использует грамматического описания языка, а основан на связывании распознающих и интерпретирующих автоматов непосредственно с метамоделью. Таким образом, как и в рассмотренных выше методах описания языков предметной области, центральной (исходной) составляющей является абстрактная структура, определяющая программу – экземпляр метамодели языка *СЛОН*.

2. Теоретическая часть

2.1. Основные положения

Как уже упоминалось выше, язык программирования (или язык предметной области, или проблемно-ориентированный язык) является средством общения с компьютером: с его помощью можно управлять действиями вычислительной машины. В этом смысле языковый процессор (реализующий язык программирования) может быть обобщен до любой прикладной программы, выполняющей некоторые действия на компьютере исходя из указаний пользователя. При этом пользователь не пишет законченную программу, а его команды интерпретируются сразу.

Например, можно рассматривать программу *Microsoft Word* как язык предметной области «Документ», с конкретным синтаксисом в виде нажиманий кнопок, заполнений форм и выборов пунктов меню, а макросы в *Microsoft Word* – как способ записи программ [18].

Таким образом, разработка любой прикладной программы по сути является разработкой проблемно-ориентированного языка.

С другой стороны, как и любой другой язык, язык программирования – это средство общения между людьми: общеизвестно, например, что хорошо написанная программа не требует комментариев и пояснений для своего понимания. Более того, язык программирования используется для выражения решения задач – то есть как средство мышления. Именно поэтому очень важна понятийная структура языка: в ее терминах мыслит человек при решении задачи.

Возвращаясь к примеру с *Microsoft Word*, можно сказать, что пользователь этой программы мыслит в терминах возможного форматирования, которое можно применить к документу, различных объектов, которые можно в него вставить, шаблонов, ссылок и так далее. А для передачи опыта другим пользователям используется, например, нотация, приведенная на рис. 2.1.1.

➢ Вставка ➢ Дата и время...
⇩ Форматы ⇗ 15-Мая-1995
☒ Вставить как поле
❑ OK

Рис. 2.1.1.

Пример формальной системы обозначений действий пользователя за компьютером [18]

Понятийная структура языка программирования определяется его абстрактным синтаксисом. При проектировании прикладной программы аналогичная структура основных сущностей создается в виде диаграммы классов *UML*.

Интерпретация языка программирования человеком и компьютером определяется его конкретным синтаксисом и семантикой. При этом с точки зрения пользователя, конкретный синтаксис языка – это его нотация, определяемая для всех понятий языка и для всех возможных действий над ними. А с точки зрения компьютера, конкретный синтаксис – это способ редактировать и транслировать программу в экземпляр метамодели. Следовательно, конкретный синтаксис может быть задан в виде редактора-транслятора, использующего соответствующую нотацию.

Семантика языка программирования определяет поведение программ, алгоритм их работы. Диаграммы состояний в *UML* – это один из методов описания алгоритмов и моделирования поведения приложений. Кроме того диаграммы состояний могут интерпретироваться не только человеком, но и компьютером (с помощью машины автоматного программирования). Следовательно, определенная с помощью диаграмм состояний семантика языка одновременно является интерпретатором данного языка.

Таким образом, метод определения языков программирования с помощью диаграмм классов и интерпретируемых автоматов позволяет использовать теоретическую спецификацию языка как его практическую реализацию за счет применения стандартных средств моделирования прикладных программ.

2.2. Определение абстрактного синтаксиса языка программирования с помощью диаграммы классов

Будем называть в дальнейшем абстрактный синтаксис языка программирования, представленный в виде диаграммы классов *UML*, метамоделью языка программирования. Тогда, класс языков, которые могут быть описаны таким образом, или их метаметамодель определяется абстрактным синтаксисом (метамоделью) диаграммы классов *UML* (например, книгу [19]).

Рассмотрим более подробно использование выразительных возможностей нотации диаграмм классов *UML* для определения метамодели языка программирования.

2.2.1. Неформальный анализ диаграммы классов как метамодели проблемно-ориентированного языка

Диаграмма классов является основным средством моделирования структуры в *UML* [20]. Для определения метамодели языка применяется один основной тип сущностей: классы, между которыми устанавливаются следующие основные типы отношений: ассоциация, обобщение, зависимости.

Аналогично тому, как формальная грамматика некоторого языка определяет синтаксическую структуру программы на этом языке и фраз и выражений, из которых она состоит, метамодель определяет абстрактную структуру программы и составляющих ее частей. При этом аналогом нетерминалов являются классы, а порождающим правилам соответствуют классы, связанные отношениями друг с другом.

Рассмотрим пример такой продукции некоторого языка:

`<Условный оператор> ::= if (<Условное выражение>) <Оператор> else <Оператор>`

Здесь жирным шрифтом выделены терминалы языка. Очевидно, что с точки зрения абстрактного синтаксиса, это правило означает, что любой условный оператор состоит из условного выражения и двух операторов (каким образом такая абстрактная фраза будет обрабатываться – это уже вопрос семантики языка). Соответственно в метамодели языка необходим класс **Условный оператор**, связанный отношениями ассоциации с классами **Условное выражение** и **Оператор** (рис. 2.2.1.1). При этом роли полюсов ассоциаций (else и then) передают семантический смысл этих отношений в рассматриваемой конструкции языка.

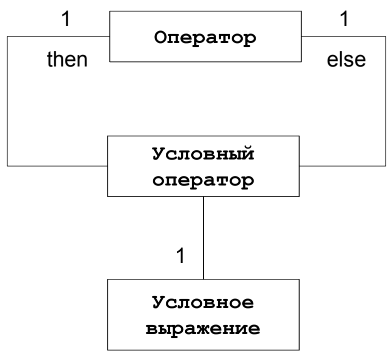


Рис. 2.2.1.1.

Фрагмент метамодели для условного оператора

Но условный оператор является частным случаем оператора, также как и, например, оператор присваивания. В грамматике языка этот факт обычно описывается с помощью альтернативных продукций, например:

`<Оператор> ::= <Условный оператор> | <Оператор присваивания>`

В диаграмме классов *UML* для этого существует отношение обобщения: класс **Оператор** является суперклассом для классов **Условный оператор** и **Оператор присваивания** (рис. 2.2.1.2).

Рис. 2.2.1.2.

Фрагмент метамодели с отношением обобщения

Таким образом, отношение обобщения, используя принцип подстановочности, предоставляет мощный механизм рекуррентных определений. Операция конкретизации позволяет построить конкретную программу (экземпляр метамодели) в силу того, что обобщения в модели образуют строгий частичный порядок (такое ограничение накладывается на них в диаграмме классов *UML*). Такая программа является, вообще говоря, графом, у которого экземпляры классов – суть вершины, а экземпляры отношений ассоциаций – суть ребра.

Из приведенного выше примера видно, что выразительные средства диаграммы классов *UML* могут заменить все выразительные средства формальных грамматик:

1. Множество нетерминалов заменяется множеством классов.
2. Множество терминалов разделяется на «семантически значимые» токены (например, имена и значения переменных, знаки операций) и «разделители» (скобки, запятые и т.д.). Семантически значимые токены заменяются значениями атрибутов соответствующих классов (например, идентификатор переменной – это атрибут класса **Переменная**), а разделители в метамодели не указываются.
3. Множество правил заменяется отношениями между классами.
4. В качестве начального нетерминала выступает класс, определяющий абстрактную программу.
5. Отношение выводимости заменяется отношением конкретизации.

Определим более формально некоторые введенные здесь термины и правила замены (сведения) формальной грамматики к метамодели. Будем говорить, что объект *a* и класс *A* связаны отношением конкретизации, если объект *a* является экземпляром класса *A*.

Пусть дано правило контекстно-свободной грамматики языка: `A ::= α`. Здесь *A* – это нетерминал, а *α* – это цепочка, состоящая из терминалов и нетерминалов. Тогда в

метамодели определяется класс *A* и классы, соответствующие всем нетерминалам из цепочки α, которые затем связываются с классом *A* отношениями ассоциации. Для семантически значимых терминалов из цепочки α вводятся атрибуты в классе *A*, а разделители отбрасываются.

Пусть теперь дан набор альтернативных продукций контекстно-свободной грамматики: `A ::= B1 | B2 | … | Bn`. Здесь предполагается, что для всех альтернативных цепочек, которые могут быть получены из *A*, вводятся вспомогательные нетерминалы B_i (такое преобразование тривиально). В этом случае определяется суперкласс *A* и классы B_i, которые он обобщает.

В качестве формальной грамматики берется контекстно-свободная грамматика, так как она является на данный момент наиболее теоретически обоснованным и алгоритмически подкрепленным методом описания формальных языков [7]. Приведенные сведения правил грамматики к фрагментам метамодели не постулируются как метод определения классов, их атрибутов и отношений. Эти замены рассматриваются только с целью продемонстрировать, что выразительные средства диаграммы классов не хуже выразительных средств формальных грамматик. Разработка метамодели языка (как и любой другой модели в диаграмме классов) относится к прагматике программирования и на данный момент (насколько известно автору) не подкреплена формальной теорией, позволяющей построить единственно правильную и работающую модель.

С помощью классов и отношений между ними (ассоциаций и обобщений) определяется абстрактный синтаксис языка *СЛОН*.

Кроме рассмотренных основных выразительных средств диаграммы классов, для определения метамодели языка могут использоваться и другие элементы этой нотации *UML*: агрегации, композиции, кратность и роли полюсов, зависимости.

2.2.2. Метамодель языка *СЛОН*

Рассмотрим фрагмент метамодели языка *СЛОН*: определение табличных выражений. Как уже упоминалось ранее, в языке *СЛОН* таблица – это по сути итератор, перебирающий элементы некоторой конечной последовательности кортежей. Табличные выражения реализуют механизм алгебры таблиц, позволяющий строить из имеющихся таблиц произвольные таблицы и выполнять над их элементами определенные действия [5]. Таким образом, табличные выражения позволяют описать в лаконичной форме нетривиальную семантику предметной области. Поэтому, именно этот фрагмент языка *СЛОН* наиболее интересен с точки зрения анализа рассматриваемого в данной работе метода.

Для таблиц определены следующие операции: сложение, умножение (бинарные), итерация, проекция, выборка (в этих операция участвует только одна таблица, поэтому будем называть их таблично унарными, или унарными). Результатом применения любой из этих операций является таблица. Кроме того, таблица-итератор перебирает кортежи «не впустую»: над каждым кортежем может выполняться некоторое действие. Если обозначить такой процесс термином «выполнение таблицы», то перед выполнением таблицы и после него могут также быть выполнены некоторые действия. Как действия над каждым кортежем, так и действия до и после выполнения таблицы являются атрибутами (свойствами) таблицы.

Исходные таблицы (с помощью которых строится табличные выражения) – это либо именованные таблицы, либо изображение таблицы. Именованные таблицы хранятся в архиве системы *ЭРА*. Изображение таблицы – это непосредственное задание таблицы в виде ее заголовка и перечисления значений всех ее кортежей.

Как уже упоминалось выше значение результата табличной операции – это тоже таблица. С другой стороны табличная операция осуществляется над таблицами. Поэтому, определяются классы **TableOp** (табличная операция) и **TableExpr** (табличное выражение) и соответствующие отношения между ними (рис. 2.2.2.1). Однако унарные операции могут быть представлены не только в виде табличной операции, но и как свойства (атрибуты) табличного выражения. Такое представление позволит жестко задать кратность полюса отношения композиции, не загромождая диаграмму дополнительными классами (например, классом унарной операции). В результате, после добавления классов **NamedTable** (именованная таблица) и **Denotation** (изображение) и действий как атрибутов таблиц, получается фрагмент языка *СЛОН*, определяющий табличные выражения (рис. 2.2.2.2).

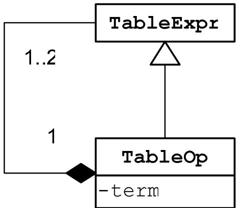


Рис. 2.2.2.1.

Отношения между классами **Table** и **TableExpr**

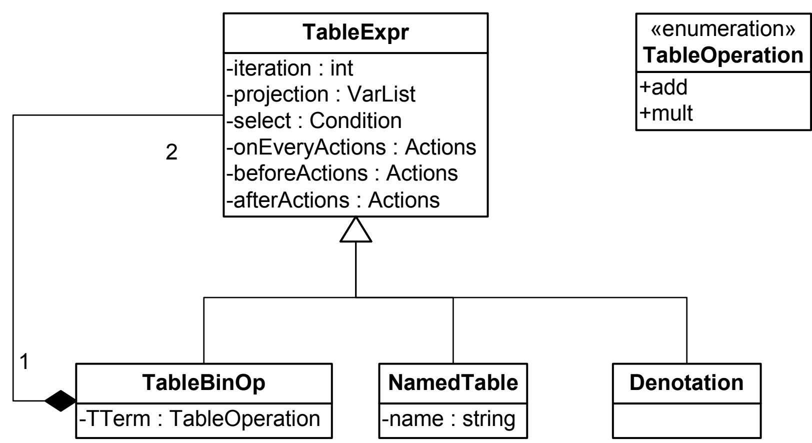


Рис. 2.2.2.2.

Фрагмент метамодели языка СЛОН: табличное выражение

На рис. 2.2.2.3 приведен пример экземпляра метамодели, соответствующего табличному выражению: `T1 + T2 * T3`. Здесь T1, T2, T3 – это идентификаторы именованных таблиц, а символы '+' и '*' означают табличные операции сложения и умножения соответственно (согласно конкретному синтаксису *СЛОНА* операция умножения приоритетнее операции сложения).

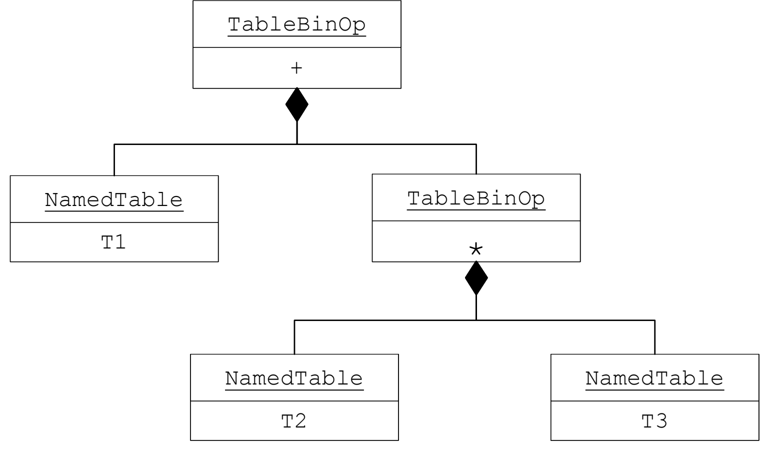


Рис. 2.2.2.3.

Дерево табличного выражения T1 + T2 * T3

В языке *СЛОН* каждой таблице в табличном выражении соответствует отдельный итератор. Следовательно, будем считать, что конкретные табличные выражения в языке *СЛОН* могут быть только деревьями. То есть, если именованная таблица встречается в табличном выражении более одного раза, то в экземпляре метамодели каждому ее вхождению соответствует отдельный объект.

2.3. Определение конкретного синтаксиса языка программирования с помощью распознающих автоматов

2.3.1. Использование парадигмы автоматного программирования

Парадигма автоматного программирования развивается и успешно применяется [0]. Она основывается на представлении и реализации программы в виде совокупности объектов предметной области и управляющих ими конечных автоматов. У объектов управления есть набор некоторых методов для выполнения, и они могут генерировать некоторые события. Автоматы при поступлении событий вызывают соответствующие методы объектов. В статье [21] рассматривается *использование автоматного программирования* для создания системы автоматического завершения ввода.

В контексте данной работы очень важен предлагаемый авторами этой статьи метод описания конкретного синтаксиса языка с помощью диаграммы состояний, без использования формальных грамматик. Он основан на преобразовании набора диаграмм состояний, определенных для каждого правила вывода *LL(1)* грамматики, в одну диаграмму состояний. Описание правил вывода формальной грамматики с помощью диаграмм переходов – это стандартный подход, используемый в классической теории трансляции (например, [3] и [7]). В ходе преобразования диаграмм осуществляется удаление правой рекурсии, удаление немотивированных переходов, подстановка диаграмм состояний друг в друга и удаление срединной рекурсии. В результате получается одна диаграмма состояний без упоминания нетерминалов на ней. Следовательно, эта диаграмма может быть рассмотрена как конечный автомат, реагирующий на терминалы – события, поставляемые ему лексическим анализатором. То есть, с точки зрения парадигмы автоматного программирования, такая диаграмма состояний – это синтаксический анализатор.

На рис. 2.3.1.1 и 2.3.1.2 приведены примеры соответственно набора исходных диаграмм и полученной в результате диаграммы состояний. Описанный подход используется авторами статьи [21] для создания системы автоматического завершения ввода: при подаче на вход такому синтаксическому анализатору незавершенной строки, автомат останавливается в каком-то состоянии. События, заданные на переходах из этого состояния, определяют множество терминалов, которые могут продолжить незавершенную строку согласно синтаксису языка. Эта система реализована в очередной версии пакета *UniMod* [22] с помощью его предыдущей версии.

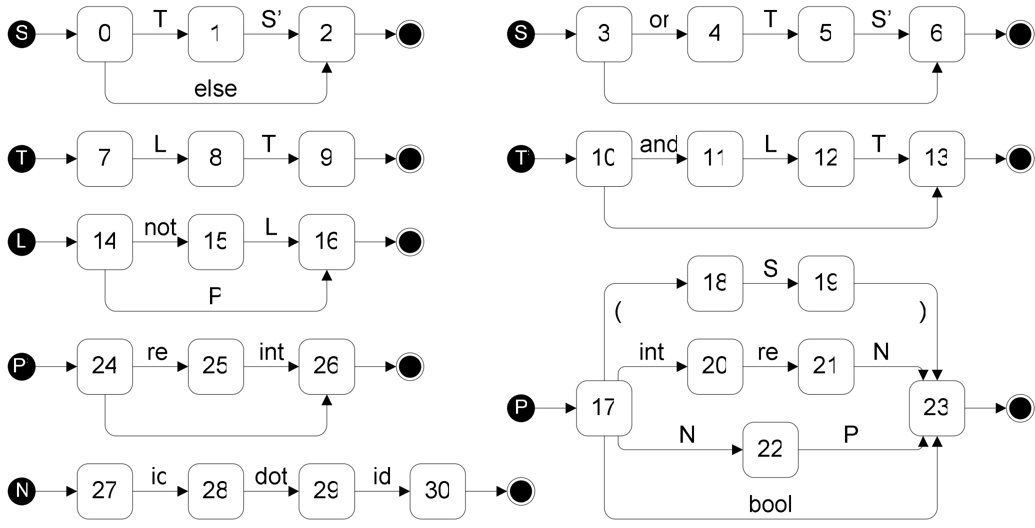


Рис. 2.3.1.1.

Набор диаграмм состояний для правил вывода некоторой грамматики (S, S', T, T', L, P, P', N - нетерминалы) [21]

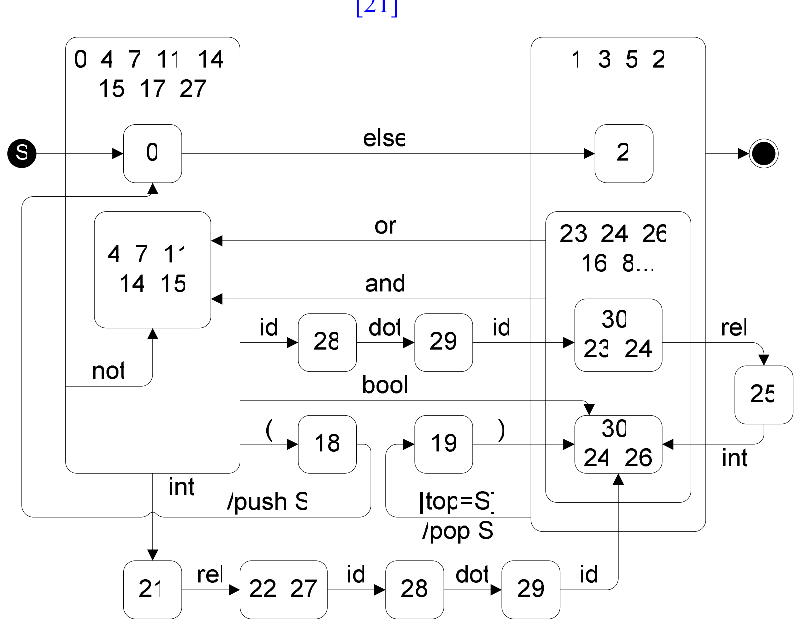


Рис. 2.3.1.2.

Полученная в результате преобразований диаграмма (остался только начальный нетерминал S) [21]

Таким образом, диаграмма состояний, содержащая на переходах терминалы языка, является реализующей язык программой (так как есть виртуальная машина автоматного программирования). С другой стороны, эта диаграмма формально определяет конкретный синтаксис языка.

Согласно классическому определению, на рис. 2.3.1.2 изображен, вообще говоря, магазинный автомат, множество входных символов которого – это множество терминалов. Последний шаг преобразования – удаление срединной рекурсии – основан на явном использовании стека (или магазина). Действительно, в книге [3] приведена теорема, согласно которой каждой системе конечных автоматов соответствует магазинный автомат, допускающий то же множество цепочек, что и эта система конечных автоматов

(теорема 3.2.1). Однако описание всего языка с помощью одной диаграммы состояний трудоёмко и тяжело для понимания. Система конечных автоматов намного удобнее для создания, восприятия и модификации (как и любая реализация принципа модульности). Использование магазина также вносит дополнительную сложность.

Исходя из всех описанных выше рассуждений, конкретный синтаксис проблемно-ориентированного языка определяется в виде системы конечных автоматов. Для этого предлагается использовать следующий метод:

1. Входным алфавитом системы автоматов является множество терминалов языка;
2. Для описания взаимодействия автоматов используется нотация диаграммы состояний *UML*: составное состояние – это переход к соответствующему автомату.

При этом под терминалом понимается элемент нотации языка или с практической точки зрения событие, посылаемое синтаксическим анализатором текста или графическим редактором, или диалоговым окном – любым источником событий, что позволяет использовать не только текстовое представление программы. Далее покажем, как из метамодели языка можно получить прототипы распознающих конечных автоматов.

2.3.2. Система автоматов из метамодели: абстрактный конкретный синтаксис

Метамодель описывает абстрактную структуру языка, для которой затем задается конкретное представление. Поэтому структура анализатора этого представления может определяться метамоделью. Будем строить анализатор некоторого представления языка СЛОН, определяемого множеством абстрактных терминалов Θ, в виде системы автоматов. Заметим, что основанный на автоматах анализатор разбирает программу последовательно. Следовательно, программа (*P*) – это конечная последовательность терминалов:

$$P = (x_1, x_2, \dots x_n), \text{ где } x_i \in \Theta$$

Аналогично тому, как диаграммы состояний в рассмотренном выше методе определялись для каждого нетерминала, можно предположить, что следует построить автомат-анализатор для каждой сущности (каждого класса) метамодели. При этом следует учитывать, что анализируется конкретная программа – экземпляр метамодели.

Согласно описанному выше фрагменту метамодели языка *СЛОН*, табличное выражение конкретизируется либо именованной таблицей, либо изображением таблицы, либо табличной операцией. Следовательно, автомат-анализатор для сущности TableExpr содержит ветвление в автоматы-анализаторы этих сущностей. Кроме того, этот автомат содержит анализ всех атрибутов класса TableExpr.

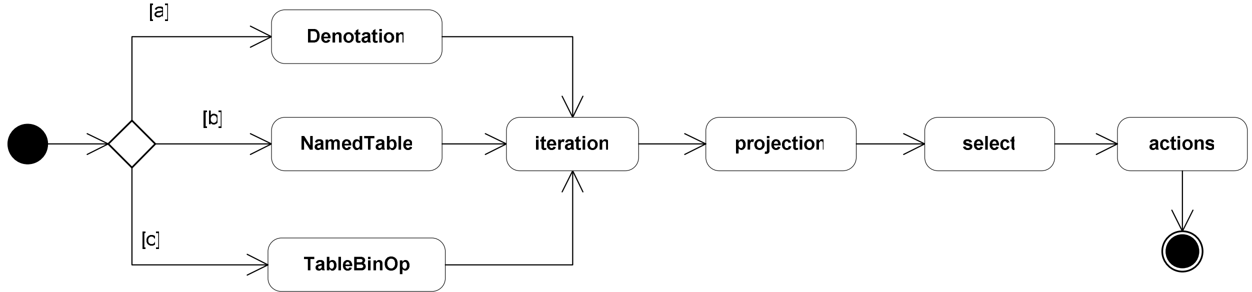


Рис. 2.3.2.1.

Автомат-анализатор для сущности TableExpr

На рис. 2.3.2.1 все состояния – составные: каждое из них означает переход к автомату-анализатору соответствующей сущности (класса) или ее свойства (атрибута класса). Все эти автоматы, кроме TableBinOp, определяются конкретным синтаксисом, также как и их последовательность вхождения в автомате TableExpr. Ключевым моментом для рассматриваемого прототипа автомата является только обязательное различие абстрактных терминалов a, b, c: это условие необходимо для корректной работы анализатора и аналогично левой факторизации грамматики.

Для TableBinOp также можно построить прототип автомата-анализатора: он реализует композицию TableBinOp и TableExpr, перебирая табличные выражения, чередующиеся знаком операции d (рис. 2.3.2.2). Здесь TableExpr – это тоже составное состояние, которому соответствует рассмотренный выше автомат (изображенный на рис. 2.3.2.1).

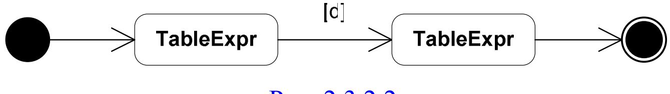


Рис. 2.3.2.2.

Автомат-анализатор для сущности TableBinOp

Таким образом, мы получили анализатор в виде системы автоматов, рекурсивно вызывающих друг друга. Начальным, или головным, автоматом для этой системы является автомат TableExpr.

<u>Докажем, что</u> если на вход такому анализатору поступает синтаксически правильная программа, то анализатор завершит свое выполнение в конечном состоянии головного автомата.

<u>Доказательство</u>

Синтаксически правильная программа определяет конечное дерево – экземпляр метамодели языка СЛОН. Для каждой сущности в этом дереве осуществляется переход во вложенный автомат. Следовательно, можно построить дерево переходов автоматов, если для каждой новой сущности в дереве программы выделить отдельный экземпляр анализирующего автомата (или совокупность экземпляров) (рис. 2.3.2.3).

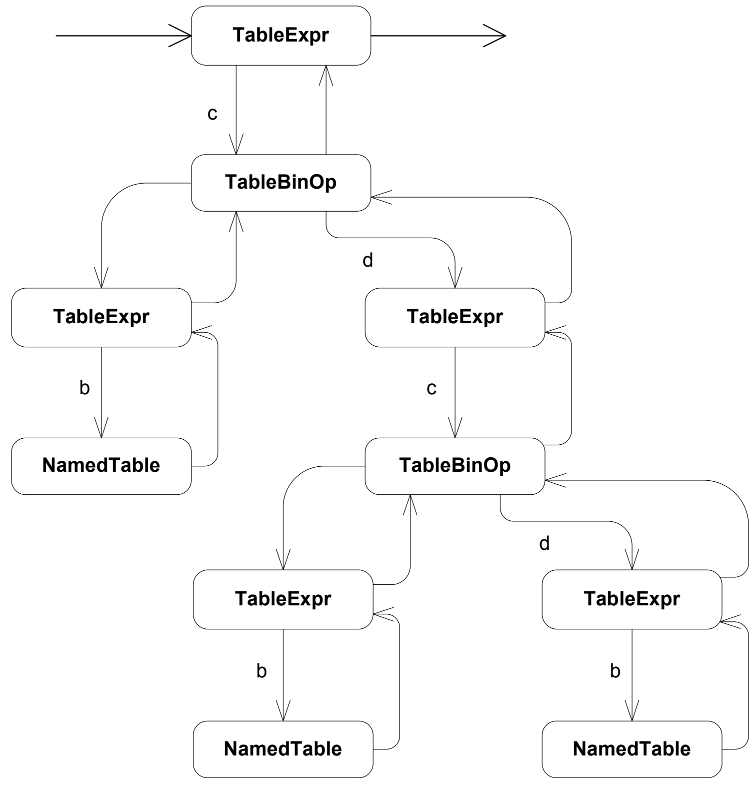


Рис. 2.3.2.3.

Дерево переходов автоматов, соответствующее конкретной программе (T1 + (T2 + T3)), или последовательности абстрактных терминалов cbdcbdb

Тогда конечное дерево программы однозначно определяет конечное дерево переходов автоматов. Выполнение алгоритма для конкретной программы – это обход конечного бинарного дерева, который завершится.

Конец доказательства.

Если же на вход анализатору поступает синтаксически неверная программа, то анализатор остановится в некотором состоянии, соответствующем синтаксически правильной подпоследовательности терминалов в начале программы.

Таким образом, из метамодели языка может быть получен прототип его анализатора. При этом отношение обобщения сводится к ветвлению в автомате, а отношение ассоциации – к циклу в автомате.

2.3.3. Конкретный синтаксис и нотация языка *СЛОН*

Конкретный синтаксис языка определяется с помощью задания множества терминалов (конкретизации абстрактных терминалов Θ) – будем называть это множество нотацией языка – и с помощью внесения необходимых изменений в структуру автоматов-анализаторов.

Нотация языка должна удовлетворять приведенным выше рассуждениям: с помощью элементов нотации можно задать программу в виде их конечной последовательности. Элементы нотации должны быть различимы.

Примером такой нотации является заполнение форм диалоговых окон: терминал – это событие, генерируемое при щелчке мышкой на кнопке. А последовательное заполнение формы – это программа.

Определим синтаксис языка *СЛОН*: терминал – это лексема, полученная из лексического анализа текстовой программы. На рис. 2.3.3.1 приведен автомат-анализатор сущности TableExpr, соответствующий конкретному синтаксису. Для того чтобы различать простые и составные состояния, последние будем выделять зеленым цветом. Лексемы отмечены синим цветом.

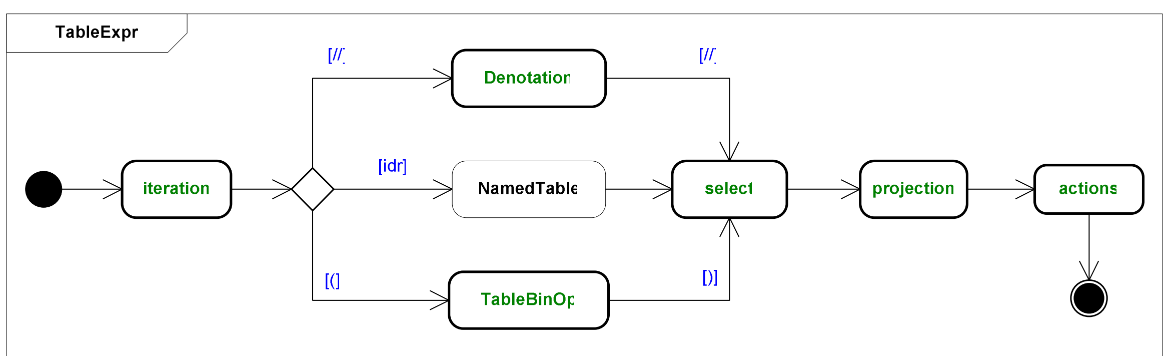


Рис. 2.3.3.1.

Конкретный синтаксис сущности TableExpr

Заметим, что составное состояние iteration теперь перед ветвлением: так конкретный синтаксис задается в структуре автомата. Это означает, что в конкретной программе в табличном выражении сначала описывается операция итерации, потом таблица, к которой она применена, а затем операции выборки, проекции и действия для этой таблицы. Кроме того, в языке *СЛОН* именованная таблица обозначается только своим идентификатором, поэтому достаточно простого (а не составного) состояния NamedTable.

Автомат-анализатор итерации позволяет опустить эту операцию с помощью условия else, которое означает, что если не поступило лексемы number, то автомат завершает свою работу (рис. 2.3.3.2).

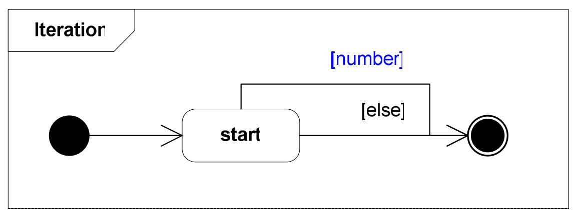


Рис. 2.3.3.2.

Конкретный синтаксис свойства Iteration

Согласно конкретному синтаксису языка *СЛОН* в табличных операциях число таблиц, следующих друг за другом через знак операции, конечно, но не ограничено (двумя таблицами). Это не противоречит метамодели языка, так как аргументы операции образуют при этом упорядоченный список, который может быть представлен в виде бинарного дерева. При этом операция умножения приоритетнее операции сложения. Обе эти особенности переданы в автомате-анализаторе сущности TableBinOp (рис. 2.3.3.3).

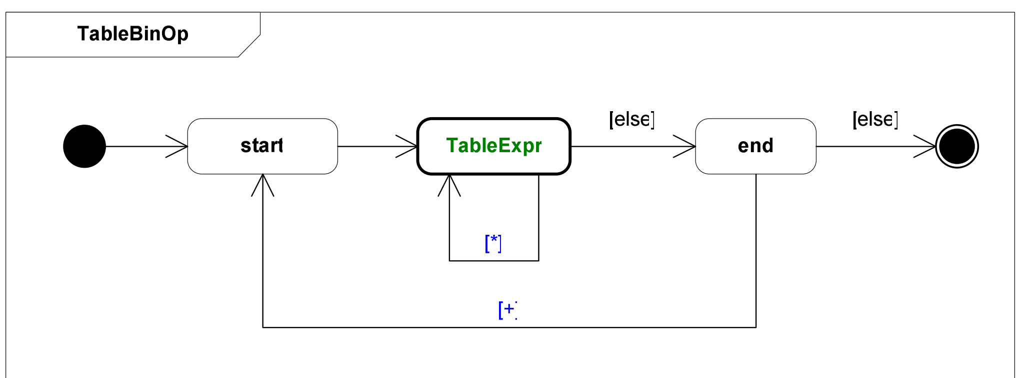


Рис. 2.3.3.3.

Конкретный синтаксис сущности TableBinOp

Заметим, что теперь в качестве начального автомата удобно выбрать TableBinOp, так как это позволит использовать табличные операции без специальной лексемы ‘(’ – сразу как последовательность таблиц, следующих через разделитель-знак операции.

Итак, конкретный синтаксис языка *СЛОН* определяется на основе прототипа системы автоматов, полученного из метамодели, с помощью задания конкретной нотации и модификации структуры автоматов.

2.4. Определение семантики языка программирования с помощью интерпретирующих автоматов

Операционный подход к определению семантики языка предполагает описание алгоритма интерпретации программы в терминах некоторой абстрактной машины. Будем строить алгоритм интерпретации программы на языке *СЛОН* в виде системы конечных автоматов. При этом интерпретируемая программа – вход алгоритма – это дерево-

экземпляр метамодели. Основной особенностью языка *СЛОН* является использование алгебры таблиц, именно табличные операции и табличные выражения представляют наибольший интерес с точки зрения описания семантики языка и поэтому рассматриваются далее в качестве примера применения предлагаемого метода.

2.4.1. Семантика табличного сложения и умножения в языке *СЛОН*

Как уже упоминалось, таблица в языке *СЛОН* – это итератор некоторой упорядоченной последовательности кортежей. Эта последовательность характеризуется заголовком таблицы (*T(A)*, где *A* – таблица) и длиной таблицы (*L(A)*).

Операция сложения таблиц определена следующим образом:

- заголовок суммы таблиц – это объединение заголовков слагаемых;
- в сумму входят сначала все кортежи первого слагаемого, а затем все кортежи второго слагаемого;
- недостающие значения в кортежах берутся по умолчанию;

Если рассматривать таблицу как программу, определяющую последовательность присваивания значений переменным (как цикл), то сложению таблиц соответствует последовательное выполнение двух циклов.

На рис. 2.4.1.1 приведен алгоритм сложения таблиц, взятый непосредственно из определения. Здесь для обозначения конца таблицы используется флаг eot («end of table»). Действие nextTuple выдает очередной кортеж таблицы. Красным цветом выделены названия таблиц-операндов, зеленым цветом обозначен вложенный автомат-интерпретатор таблицы.

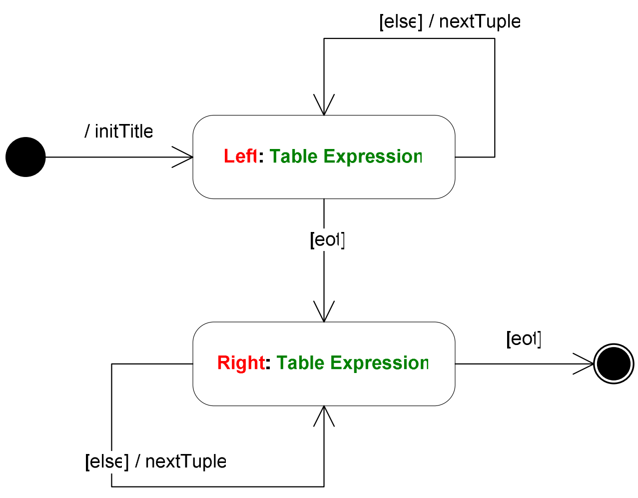


Рис. 2.4.1.1.
Операция сложения таблиц: первое приближение

Действительно, в качестве таблиц-операндов в сложении могут выступать, вообще говоря, табличные выражения, содержащие в том числе сложение таблиц. Для того чтобы

корректно выполнить всё табличное выражение, автомат должен выдавать очередной кортеж и завершаться (чтобы этот кортеж затем использовался для реализации всего табличного выражения). Соответственно, когда вновь понадобится кортеж из сложения необходимо вернуться именно в ту таблицу-операнд, на которой был совершен выход в предыдущий раз. Такая особенность описывается в диаграмме состояний *UML* с помощью исторического состояния. Модифицированный согласно этим рассуждениям автомат представлен на рис. 2.4.1.2.

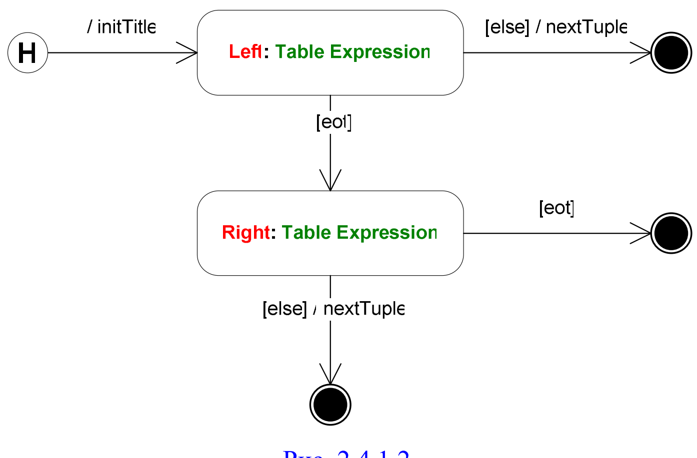


Рис. 2.4.1.2.

Операция сложения таблиц: второе приближение

Однако, в различных операциях сложения, принадлежащих одному и тому же табличному выражению, интерпретатор может оказаться в различных состояниях: например, в одной операции левая таблица уже закончилась, а в другой – еще нет. Для того чтобы избежать такой некорректности, каждому вхождению операции сложения должен соответствовать свой экземпляр автомата-интерпретатора. Таким образом, если интерпретируемая программа – это дерево, то каждому вхождению операции сложения в этом дереве соответствует узел с непустыми левым и правым поддеревьями – аргументами операции. Тогда для каждого такого узла создается экземпляр автомата сложения, который реализует таблицу-итератор, а именно: перебирает кортежи – элементы таблицы, которая получена сложением таблиц, определяемых левым и правым поддеревьями узла программы.

В автомате сложения результирующая последовательность кортежей строится из последовательностей-операндов с помощью составных состояний Left и Right. Вложенные в эти состояния автоматы реализуют необходимые последовательности кортежей на основе соответственно левого и правого поддеревьев узла-сложения. Следовательно, на входе в состояние Left нужно перейти к левому поддереву, на выходе из состояния – вернуться назад, на уровень узла-сложения. Аналогично для состояния Right. В *UML* этого используется действия на входе и действия на выходе:

```
entry / goToLeft
exit / goToParent
```

Заметим далее, что автомат сложения с одной стороны использует флаг eot, с другой стороны как реализация таблицы должен предоставлять такой же флаг вызвавшему его автомату. Для передачи параметров-флагов *UML 2.0* предоставляет удобное средство: точка выхода (exit point) – это возвращаемое автоматом значение и одновременно заключительное его состояние.

Теперь автомат выглядит так, как представлено на рис. 2.4.1.3.

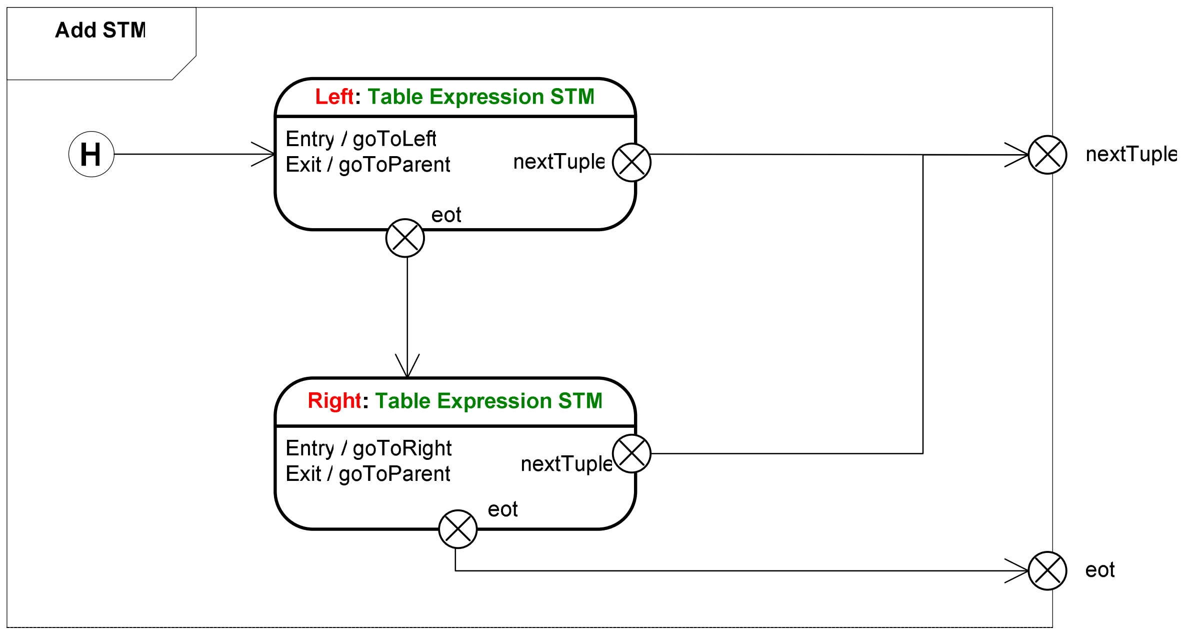


Рис. 2.4.1.3.

Операция сложения таблиц: окончательный вариант

Здесь используется принцип автоматного программирования: автомат управляет некоторым объектом, а не передает данные – непосредственно передачу кортежа автомат не выполняет. Кортежи строятся где-то в листьях дерева программы, а автомат сложения (*Add STate Machine*) только управляет процессом их построения.

Рассмотрим теперь операцию умножения таблиц. При умножении таблиц:

- заголовок произведения является объединением заголовков сомножителей (они не должны пересекаться);
- в произведение входят кортежи, образованные следующим образом: к каждому кортежу множимого по очереди приписываются все кортежи множителя.

С точки зрения таблицы-программы, умножение таблиц – это вложенность циклов. На рис. 2.4.1.4 приведен алгоритм, соответствующий этому определению.

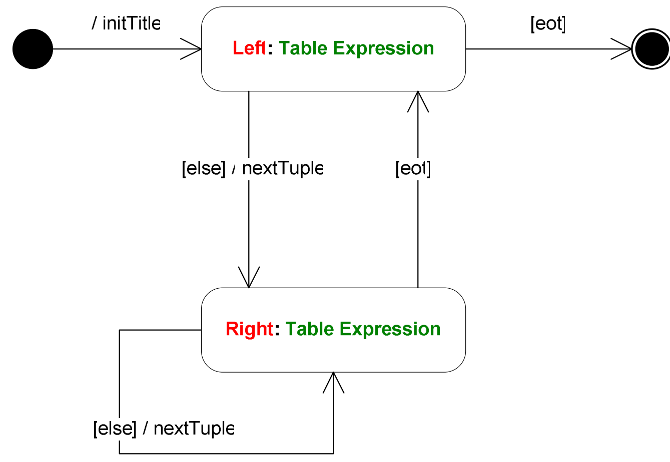


Рис. 2.4.1.4.

Операция умножения таблиц: первое приближение

После преобразований, аналогичных приведенным выше, получим автомат, изображенный на рис. 2.4.1.5.

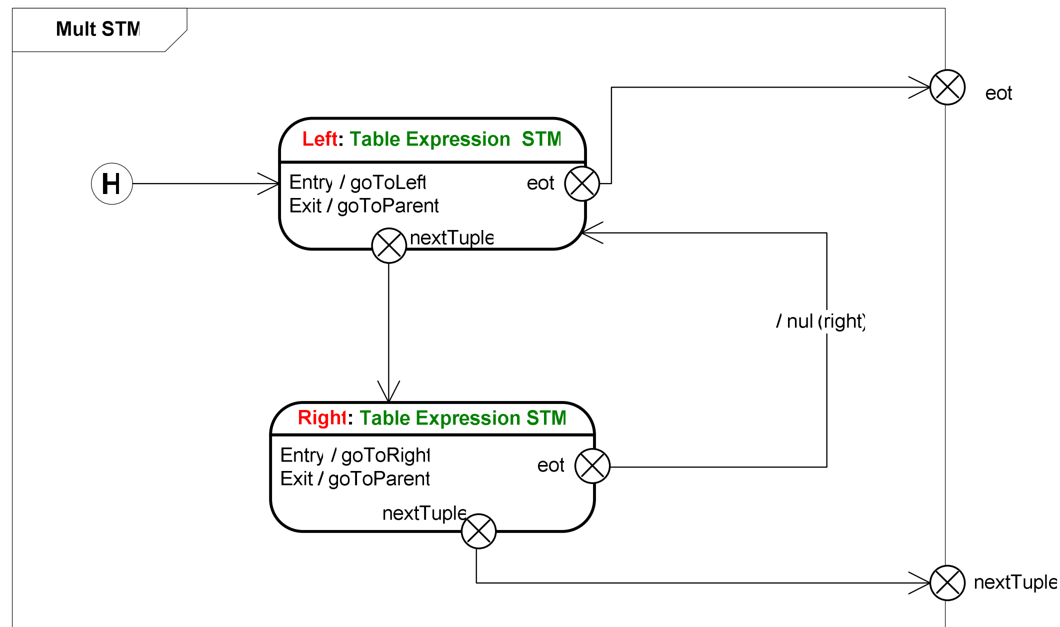


Рис. 2.4.1.5.

Операция умножения таблиц: окончательный вариант

Здесь, в отличие от автомата сложения, понадобилось обнуление экземпляра автомата, вложенного в состояние Right (null(right)). Это объясняется тем, что для каждого следующего кортежа множимого (Left) необходимо «перематывать» таблицу множителя (Right) в начало. Это равносильно обнулению соответствующего экземпляра автомата.

Таким образом, удалось описать семантику достаточно нетривиальных табличных операций с помощью лаконичных и понятных автоматов (понятных не только человеку,

но и, как убедимся в дальнейшем, *виртуальной машине автоматного программирования*). Теперь эти автоматы могут быть использованы для интерпретации табличных выражений.

2.4.2. Семантика табличных выражений языка *СЛОН*

Помимо бинарных операций, семантика табличного выражения включает таблично унарные операции и действия до, после выполнения таблицы и над каждым кортежем.

Итерация таблицы – это умножение таблицы на скаляр (натуральное число N), то есть N-кратное повторение (сложение) кортежей исходной таблицы. Так как в этой операции участвует только одна таблица, то это унарная табличная операция, представленная в метамодели в виде атрибута класса TableExpr: числа iteration (которое по умолчанию, очевидно, равно единице). Заголовок таблицы-итерации тот же, что и заголовок исходной таблицы.

Проекция таблицы определяется списком имен переменных, соответствующих столбцам исходной таблицы, которые останутся в результирующей таблице. Таким образом, в результате проекции таблицы часть ее столбцов отбрасывается. Соответственно часть заголовка – тоже. Проекция представлена в метамодели в виде атрибута projection табличного выражения, который содержит список имен переменных (по умолчанию, очевидно, это весь заголовок таблицы).

Выборка из таблицы по условию формирует результирующую таблицу из кортежей исходной таблицы, которые удовлетворяют заданному условию. Соответственно заголовок таблицы не меняется, а длина может стать меньше. Выборка представлена в метамодели атрибутом select.

Действия могут быть любыми (арифметические операторы, печать кортежа): все возможности определяются предметной областью и задаются с помощью описания конфигурации системы. Действия могут выполняться до перебора кортежей (например, чтобы проинициализировать необходимые переменные), над каждым кортежем и после их перебора. В метамодели языка они представлены соответственно атрибутами beforeActions, onEveryActions и afterActions.

С точки зрения таблицы-программы (или, точнее таблицы-итератора, определяющей заголовок цикла), итерация – это обычный цикл со счетчиком, телом которого является таблица. Проекция – это локализация переменных, выборка – условный оператор в теле цикла таблицы. Действия над каждым кортежем – вызов процедуры в теле цикла. Действия до выполнения таблицы – вызов процедуры перед циклом таблицы, действия после – соответственно после цикла таблицы.

Соответственно автомат-интерпретатор табличного выражения реализует семантику всех этих операций. Кроме того, внешне этот автомат реализует таблицу: как и автоматы сложения и умножения, экземпляр автомата табличного выражения соответствует узлу дерева программы. Следовательно, у этого узла может быть родитель-бинарная табличная операция, которой соответствует экземпляр автомата этой операции, и этот автомат должен «знать», когда готов очередной кортеж или закончилась таблица.

Следовательно, аналогично автоматам бинарных операций, автомат табличного выражения использует историческое состояние и точки выхода (рис. 2.4.2.1). Историческое состояние позволяет вернуться именно туда, где была прервана работа автомата: в выполнение таблицы, или, например, к следующей итерации таблицы.

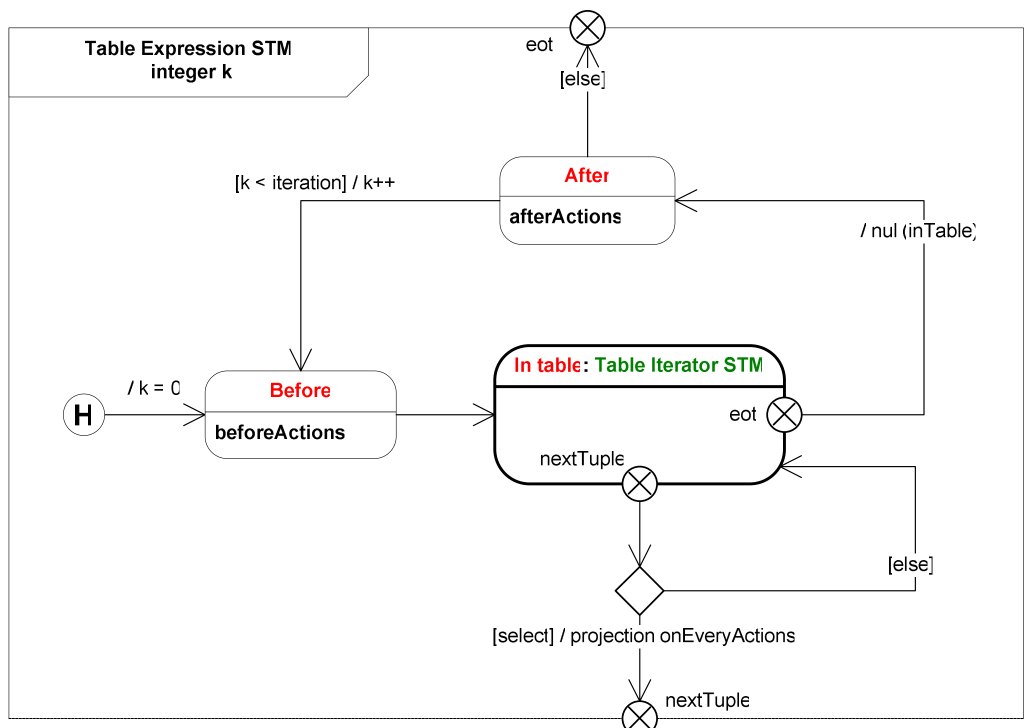


Рис. 2.4.2.1.

Автомат-интерпретатор табличного выражения

Здесь используется локальная переменная автомата **k** – для подсчета числа итераций таблицы. In table – это составное состояние, в нем выполняется конкретизация таблицы в зависимости от того является ли это табличное выражение бинарной операцией, именованной таблицей или изображением таблицы. Соответственно, это состояние реализуется вложенным автоматом Table Iterator STM. Если таблица еще не кончилась и выдала очередной кортеж (точка выхода nextTuple), то для этого кортежа проверяется условие выборки (select) и если условие выполнено, то над ним выполняется проекция (projection) и заданные действия (onEveryActions).

Вложенный автомат Table Iterator STM осуществляет спуск по дереву переходов автоматов. Это спуск либо к листу дерева программы (именованная таблица или изображение), либо к узлу с непустыми левым и правым поддеревьями (операция сложения или умножения). Этот автомат представлен на рис. 2.4.2.2.

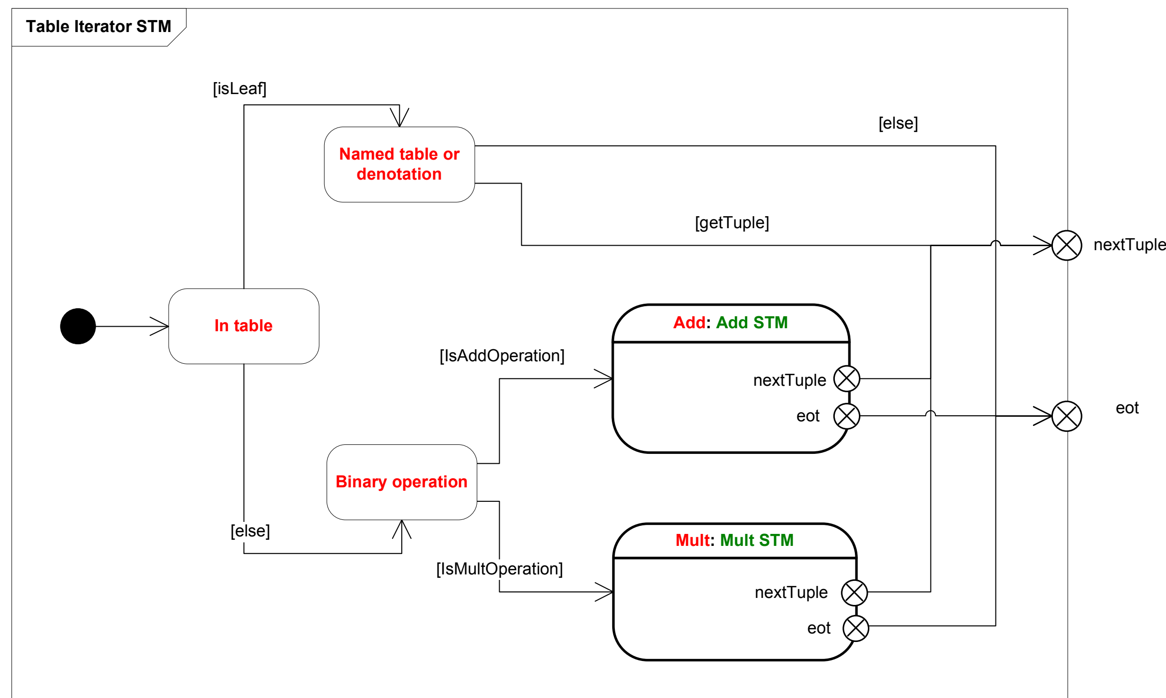


Рис. 2.4.2.2.

Вложенный автомат конкретизации таблицы

Заметим, что на приведенных автоматах используются только вызовы методов объекта управления: iteration, projection, isAddOperation и т.д. В данном случае объектом управления является дерево программы на языке *СЛОН*. А его методы реализуют непосредственно получение кортежа таблиц-операндов, формирование результирующего кортежа и выполнение операций над ним. Таким образом, автомат только управляет этим процессом: например, определяет точку «готовности» кортежа – когда с ним можно выполнять заданные операции. За счет такого усложнения объекта управления в автоматном программировании достигается практическое удобство использования автоматов.

2.4.3. Интерпретатор программы на языке СЛОН

Автомат табличного выражения заканчивает свою работу в точках выхода nextTuple или eot. Следовательно, он не может быть головным автоматом. На рис. 2.4.3.1 приведен головной автомат для интерпретатора табличного выражения, который является начальным автоматом полученной системы автоматов.

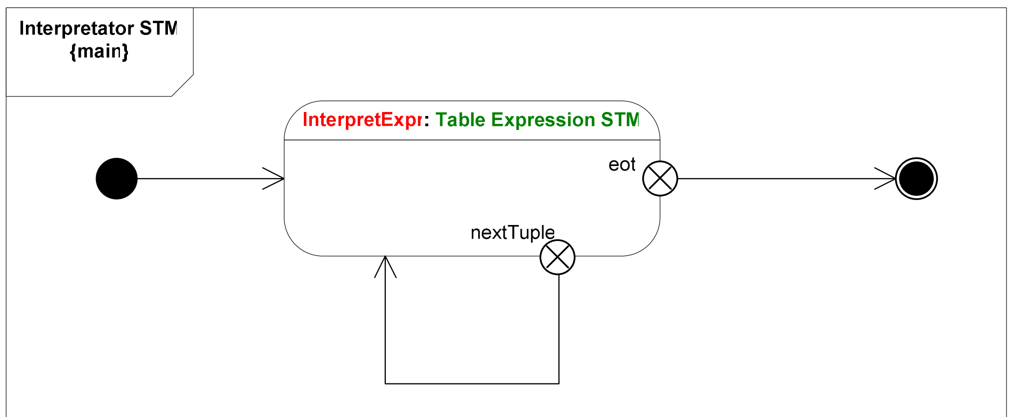


Рис. 2.4.3.1.

Головной автомат интерпретатора

Система автоматов построенных выше (Interpretator STM, Table Expression STM, Table Iterator STM, Add STM, Mult STM) реализует интерпретатор табличных выражений языка СЛОН: эти автоматы определяют семантику табличных выражений согласно своему построению.

Докажем что, если этот интерпретатор управляет синтаксически верной (согласно абстрактному синтаксису) программой, то он завершит свое выполнение в конечном состоянии головного автомата.

Доказательство

Доказательство аналогично доказательству, приведенному в разд. 2.3.2.

Синтаксически правильная программа – это конечное дерево – экземпляр метамодели языка *СЛОН*. Для каждой сущности в этом дереве осуществляется переход во вложенный экземпляр автомата Table Expression, а затем в экземпляр автомата Table Iterator. Причем для сущностей, реализующих бинарные операции (объектов класса TableBinOp) также делается переход в экземпляр автомата Add STM или Mult STM. Следовательно, всем листьям дерева программы соответствует два экземпляра автомата, а всем остальным узлам – три экземпляра автомата Дерево переходов этих экземпляров автоматов конечно.

Кроме этих экземпляров автоматов, существует также один экземпляр начального автомата, который запускает обход дерева переходов автоматов для получения каждого следующего кортежа. Число этих кортежей равно длине результирующей таблицы и поэтому конечно.

Следовательно, интерпретатор – это обход конечного дерева переходов (рис. 2.4.3.2), выполненный *L(T)* раз, где *T* – это таблица, реализуемая интерпретатором.

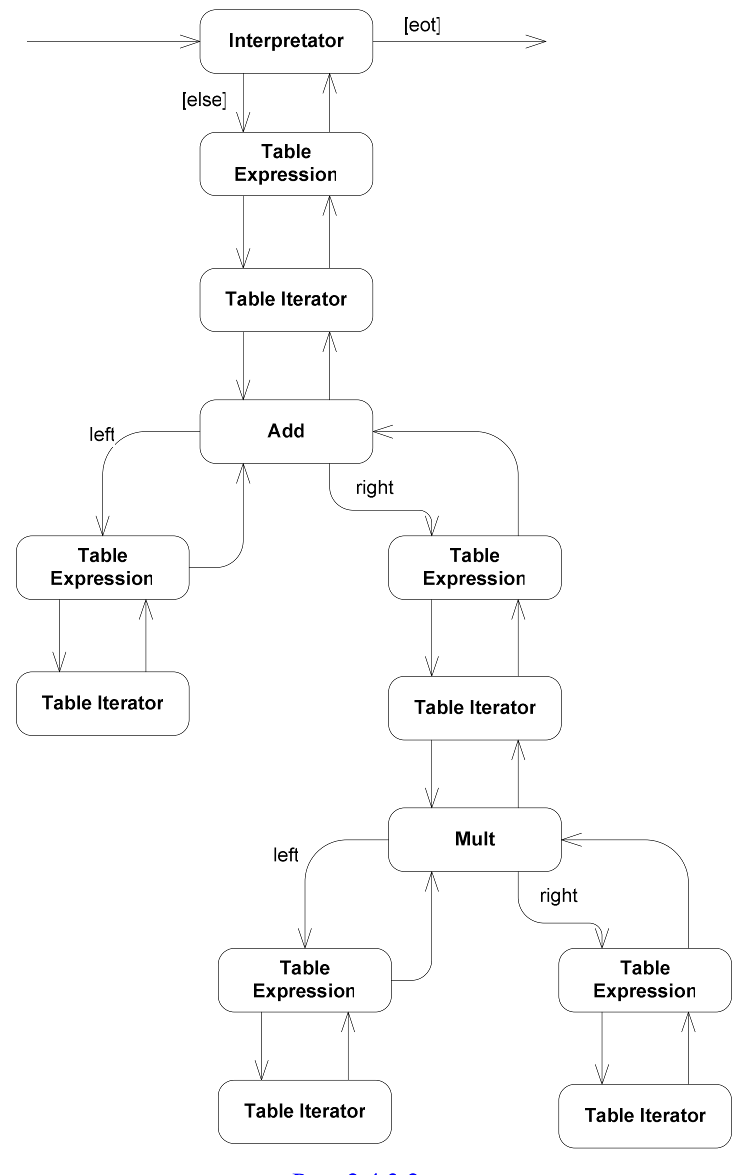


Рис. 2.4.3.2.

Дерево переходов автоматов для программы *T1* + *T2* * *T3*

Конец доказательства

Следствие

Число экземпляров автоматов интерпретатора для табличного выражения, состоящего из N исходных таблиц, определяется по формуле:

A = 5N - 2

Доказательство

N – число листьев дерева программы. Определим число **K** внутренних узлов дерева программы. Дерево программы – бинарное дерево. Пусть оно не тривиально (не один узел). Тогда, в нем каждому внутреннему узлу, кроме корневого, соответствует три ребра: для правого поддерева, для левого поддерева и к родителю. А каждому листу соответствует только одно ребро (к родителю). Корневому же узлу соответствует два ребра.

Тогда, если **q** – это число ребер дерева, то справедливо следующее соотношение:

$$2q + 1 = N + 3K$$

Кроме того, как и для любого другого дерева, для бинарного дерева программы справедливо соотношение:

$$q = p - 1 = N + K - 1$$

Из этих двух соотношений получим:

$$K = N - 1$$

Тогда число экземпляров автоматов интерпретатора равно:

$$A = 2N + 3K + 1 = 5N - 2$$

Конец доказательства

Замечание

Построенная система автоматов не использует магазин (стек) явно: он реализован по сути деревом интерпретируемой программы.

3. Проектирование

3.1. Машина автоматного программирования

3.1.1. Структура автоматной программы

Автомат как программа выполняется, используя входные события и предикаты. Результатом его выполнения являются некоторые эффекты и выходные состояния (рис. 3.1.1.1).

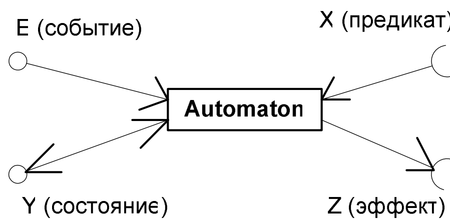


Рис. 3.1.1.1.

Внешние интерфейсы автомата

Для разработки языкового процессора языка *СЛОН*, от машины автоматного программирования не требуется реализация событийной модели: определенные выше автоматы являются, по сути, блок-схемами алгоритмов анализа или интерпретации. Поэтому, машина автоматного программирования разрабатывается без устойчивых состояний (состояний, которые ждут событий).

Автоматная программа (программа, интерпретируемая машиной автоматного программирования) специфицируется следующим образом.

Автоматная программа – это система конечных автоматов. Эти автоматы взаимодействуют друг с другом через составные состояния: во время выполнения программы экземпляр автомата, вложенного в данное составное состояния, подменяет это составное состояние. При этом переходы, которые ведут в это составное состояние, направляются в начальное состояние вложенного автомата, а переходы из составного состояния выходят из заключительных состояний этого автомата. Один из автоматов является начальным (или головным): он вызывает все остальные.

В состоянии могут выполняться некоторые побочные эффекты при попадании (**`entry`**) в него и при выходе из него (**`exit`**). Для составных состояний эффекты на входе выполняются перед переходом во вложенный автомат, а эффекты на выходе – после возврата из него.

Вложенные автоматы (то есть, все кроме начального автомата) могут начинать свое выполнение в точках входа, а заканчивать – в точках выхода. Следовательно, управление автоматом извне и получение от него некоторого результата (помимо выполнения

эффектов) осуществляется только с помощью его состояний: точка входа – это входной параметр автомата, точка выхода – возвращаемое значение.

При непосредственном выполнении экземпляр автоматной программы образует дерево переходов экземпляров автоматов: обозначим его AutoTree. Таким образом, с помощью задания конечного числа типов (классов) автоматов определяется, вообще говоря, бесконечное число конкретных автоматных программ.

Автоматы системы детерминированы. Условия на переходах задаются в виде логических формул, в которых могут использоваться функции объекта управления и локальные переменные автомата. Эффекты на переходах и в состояниях – это вызовы операций объекта управления.

Из перечисленных выше особенностей автоматной программы следует такая её структура (рис. 3.1.1.2).

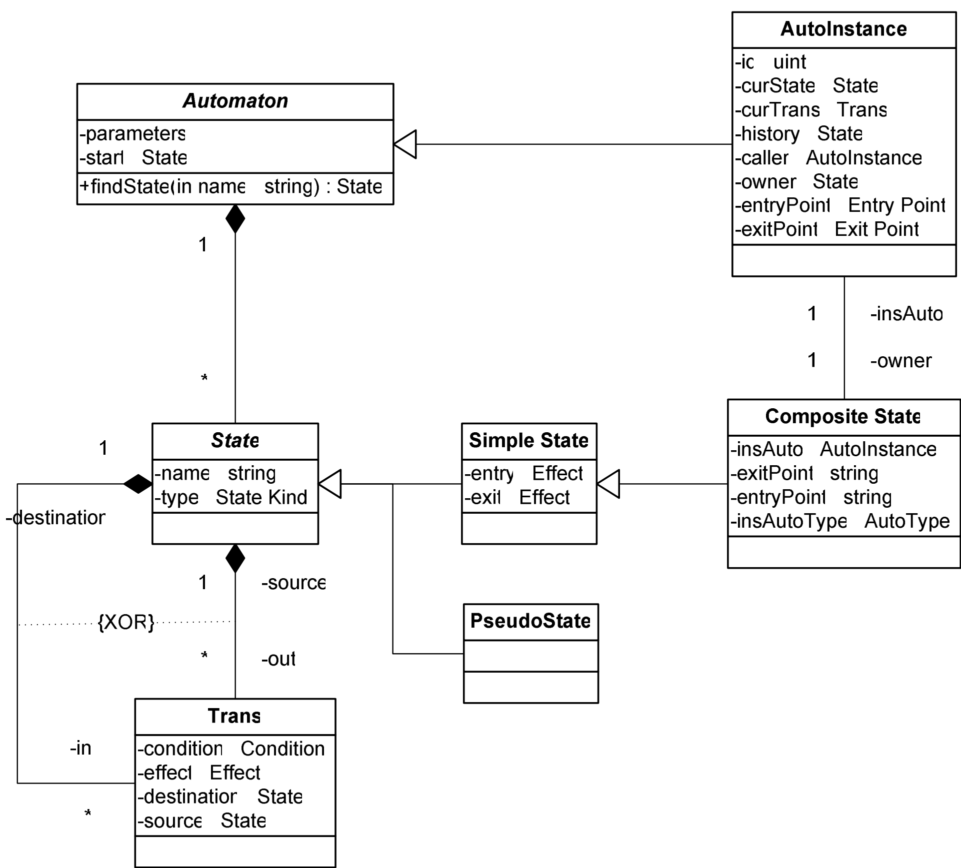


Рис. 3.1.1.2.

Структура (абстрактный синтаксис, метамодель) автоматной программы

Отдельно выделены простое состояние, составное состояние и специальное состояние. Псевдо состояния – это начальное, заключительное, историческое и точки входа и выхода. Все типы состояний идентифицируются с помощью свойства type класса *State*. Составное состояние ссылается на экземпляр автомата (insAuto), тип которого определяется вложенным автоматом (insAutoType). Точки входа и выхода являются

состояниями только для вложенного автомата. Для составного состояния, в которое вложен этот автомат, точки входа и выхода – это флаги. Соответственно, чтобы по флагу составного состояния определить точку входа вложенного автомата, вводится метод FindState класса *Automaton*.

Класс AutoInstance конкретизирует абстрактный автомат *Automaton* и в процессе выполнения сохраняет конфигурацию автомата.

3.1.2. Графический синтаксис автоматной программы

Графический синтаксис автоматной программы также как и ее структура вытекает из разработанных в теоретической части автоматов анализатора и интерпретатора языка *СЛОН*. Используемый для изображения автоматных программ графический синтаксис основан на нотации диаграмм состояний *UML*: к нему только добавлены необходимая для интерпретации конкретика и цвета – для красоты.

В табл. 1 перечислены все элементы нотации, используемой для отображения автоматных программ.

Таблица 1. Графический синтаксис автоматных программ

Элемент нотации	Пояснения
Название автомата STM [local parameters] [{main}]	Автомат описывается с помощью своего названия, списка типизированных параметров. Головной автомат помечается ключевым словом *main*
Название состояния Entry / Действия при входе Exit / Действия при выходе	Простое состояние: ключевые слова **entry** и **exit** могут быть опущены
Название состояния: название вложенного автомата STM Entry / Действия на входе Exit / Действия на выходе	Составное состояние: отличается от простого состояния только типизацией вложенного в него автомата
[условие] / действия →	Переход
●	Начальное состояние: в автомате может быть только одно начальное состояние
◉	Заключительное состояние

Ⓗ	Историческое состояние: является частным случаем начального состояния
○	Точка входа
⊗	Точка выхода
◇	Символ ветвления

Заметим, что символ ветвления не отражен в модели автоматной программы. Это сделано, потому что он введен только для упрощения схемы переходов и имеет простую семантику (рис. 3.1.2.1).

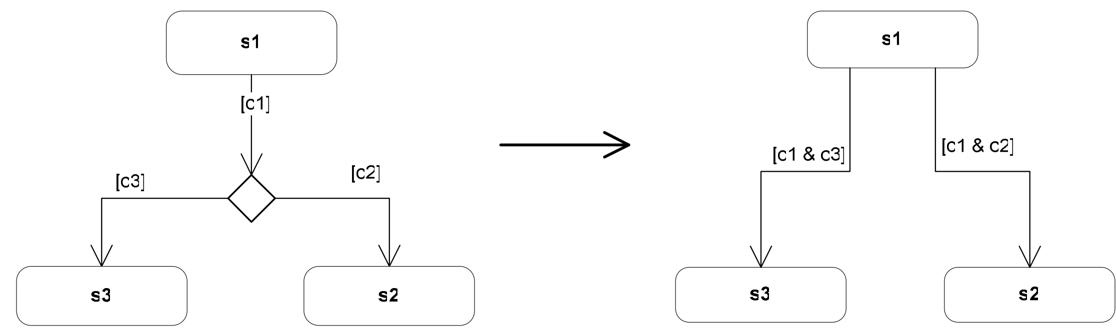


Рис. 3.1.2.1.

Семантика символа ветвления

Данное преобразование может осуществляться транслятором автоматных программ, не «засоряя» основную семантику выполнения автоматной программы.

Кроме того, модель автоматной программы рассматривает точки входа и выхода как состояния только для вложенного автомата. Для составного состояния они могут быть изображены на его границе в виде состояний, но фактически эта нотация обладает такой семантикой (рис. 3.1.2.2).

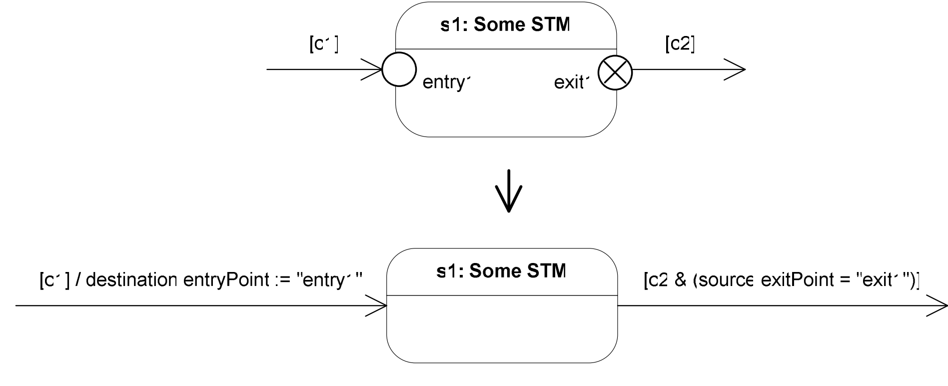


Рис. 3.1.2.2.

Семантика точек входа и выхода на границе составного состояния

3.1.3. Семантика автоматной программы

Семантика автоматной программы определяется алгоритмом её интерпретации машиной автоматного программирования (рис. 3.1.3.1).

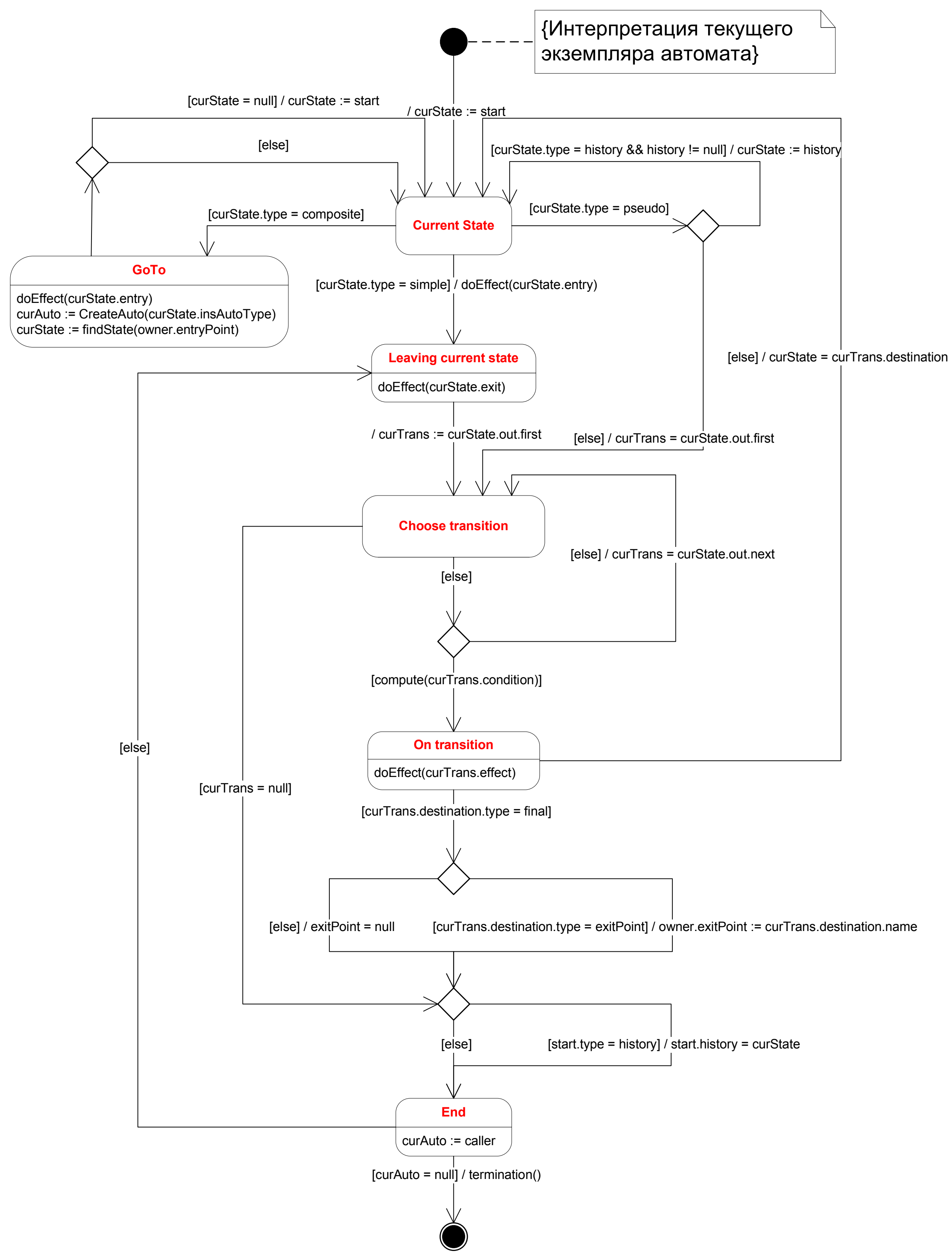


Рис. 3.1.3.1.

Алгоритм интерпретации автоматной программы

Семантика автоматной программы (и одновременно алгоритм ее интерпретации) сама является автоматной программой. Следовательно, реализован метод раскрутки.

В процессе интерпретации автоматной программы осуществляется обход дерева переходов экземпляров автоматов AutoTree. Соответственно, curAuto – это текущий экземпляр автомата. Исходно им является экземпляр начального автомата. Все действия, представленные на рис. 3.1.3.1, осуществляются над текущим экземпляром автомата.

Обработка текущего состояния зависит от его типа. В случае если это составное состояние, происходит переход во вложенный автомат. Возврат из вложенного автомата осуществляется в то же составное состояние.

Интерпретация автоматной программы основана на последовательном вычислении условий переходов (compute) и выполнении эффектов (doEffect). Эти операции реализованы с помощью блока выполнения (*ExecAuto*) на основе методов объекта управления и интерфейса машины автоматного программирования (рис.3.1.3.2). Предоставляемый машиной автоматного программирования интерфейс autoOperations позволяет влиять на выполнение автоматной программы: например, в интерпретаторе используется операция обнуления автомата.

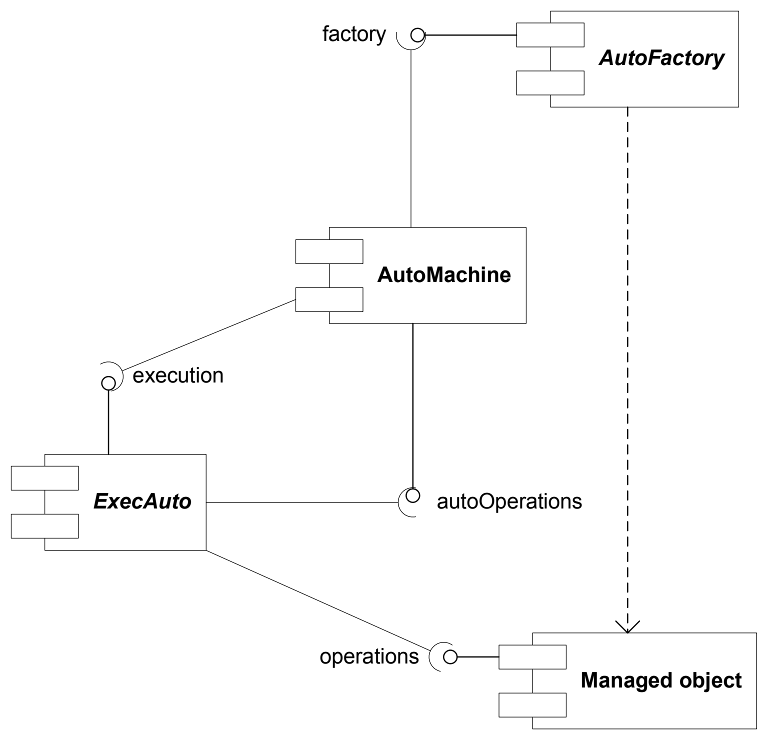


Рис. 3.1.3.2.

Диаграмма компонентов автоматной машины

Фабрика автоматов (*AutoFactory*) отвечает за создание, хранение и удаление экземпляров автоматов. При этом создаваемые автоматы связываются с интерфейсом объекта управления. На рис. 3.1.3.3 приведены интерфейсы основных компонентов.

«utility»
AutoMachine
-curAuto AutoInstance
-main AutoType
+terminate()
+interpretAuto()

«interface»
ExecAuto
+doEffect(in e : Effect)
+compute(in c : Condition) : bool

«utility»***AutoFactory***
+CreateAuto(in type AutoType) : AutoInstance

Рис. 3.1.3.3.

Интерфейсы компонентов

Компонент автоматной машины (AutoMachine) интерпретирует автоматную программу, начиная с ее головного автомата (атрибут main). Алгоритм интерпретации основан на более низком уровне абстракции выполнения, предоставляемом блоком выполнения *ExecAuto* в виде вычисления условий и исполнении эффектов. Способ реализации блока выполнения может сильно зависеть от используемой среды (аппаратного и программного обеспечения), поэтому скрывается.

3.2. Интерфейс дерева программы

Как уже отмечалось выше, парадигма автоматного программирования предполагает усложнение объекта управления с тем, чтобы достичь практического удобства использования автоматов. Для автоматов, реализующих язык *СЛОН*, объектом управления является дерево программы на языке СЛОН: обозначим его SlonTree.

С одной стороны дерево программы SlonTree как экземпляр метамодели строится с помощью анализатора (или редактора). С другой стороны анализатор использует для этого интерфейс самого дерева SlonTree. Таким образом, дерево программы SlonTree – это «обертка» для метамодели языка *СЛОН*, реализующая все необходимые интерфейсы: для анализатора – интерфейс конструирования дерева, для интерпретатора – интерфейс табличного выражения. Кроме того, очевидно, что необходим интерфейс обхода дерева (рис. 3.2.1).

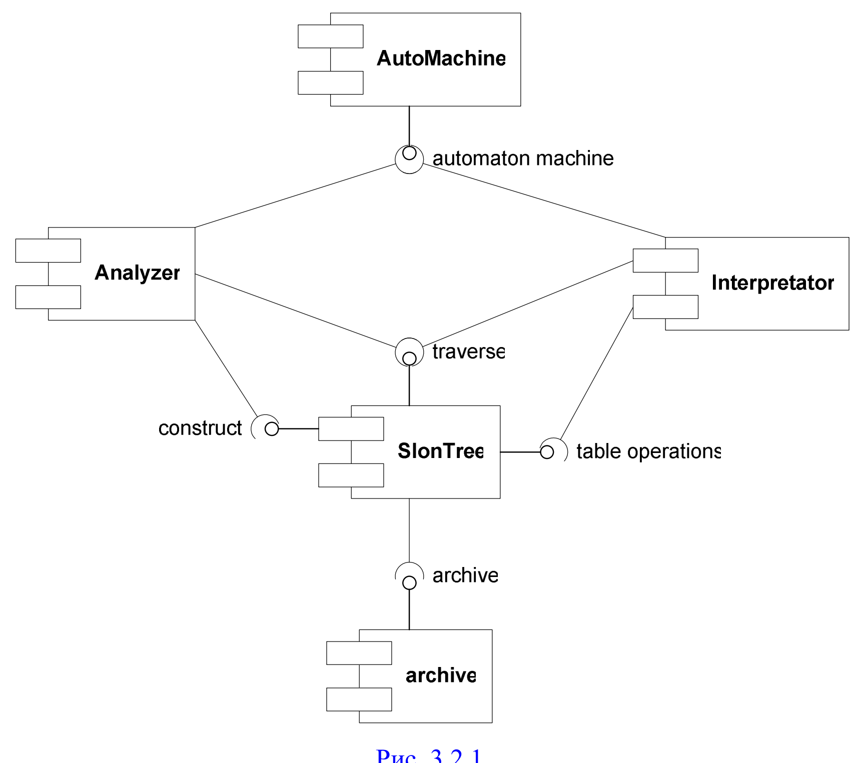


Рис. 3.2.1.

Диаграмма компонентов проектируемой системы

Дерево программы скрывает в себе работу с архивом, конфигурацией системы, формирование и обработку кортежа.

Анализ автоматов интерпретатора, определенных в предыдущей главе (рис. 2.4.1.3, 2.4.1.5, 2.4.2.1, 2.4.2.2, 2.4.3.1), позволяет определить необходимые им интерфейсы дерева (рис. 3.2.2). К функциям, использующимся как условия на переходах и выполняемые эффекты в автоматах интерпретатора, здесь добавилась функция инициализации таблицы, которая, очевидно, понадобится для подготовки заголовка таблицы и исходных таблиц выражения перед его выполнением (то есть в начале автомата Table Expression STM).

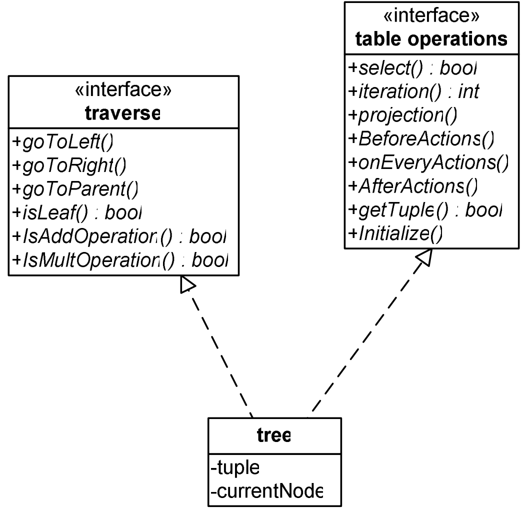


Рис. 3.2.2.

Интерфейсы дерева программы, используемые интерпретатором

Интерфейс обхода дерева программы осуществляется с помощью указателя на текущий узел в нем (currentNode), а интерфейс табличного выражения основан на хранении целевого кортежа. В процессе обхода дерева программы, этот кортеж заполняется значениями (с помощью операции *getTuple*) и обрабатывается на всех уровнях дерева (каждому узлу дерева соответствует часть кортежа, определяемая заголовком таблицы этого узла).

Заметим, что операции интерфейса дерева могут быть реализованы также с помощью автоматного программирования. Для этой реализации объекты управления будут проще. Таким образом осуществляется постепенное «погружение» от верхнего уровня абстракции к детальной реализации на основе метода раскрутки.

4. Реализация и применение машины автоматного программирования

4.1. Компонентная структура машины автоматного программирования

Машина автоматного программирования реализована в виде следующих основных компонентов: автоматная машина (AutoMachine), блок выполнения (ExecAuto), фабрика автоматов (*AutoFactory*), служба ведения протокола выполнения (или согласно общепринятому жаргону, лога) (Logger), встроенные арифметико-логические операции (ALU), программа для выполнения (Program) (рис. 4.1.1).

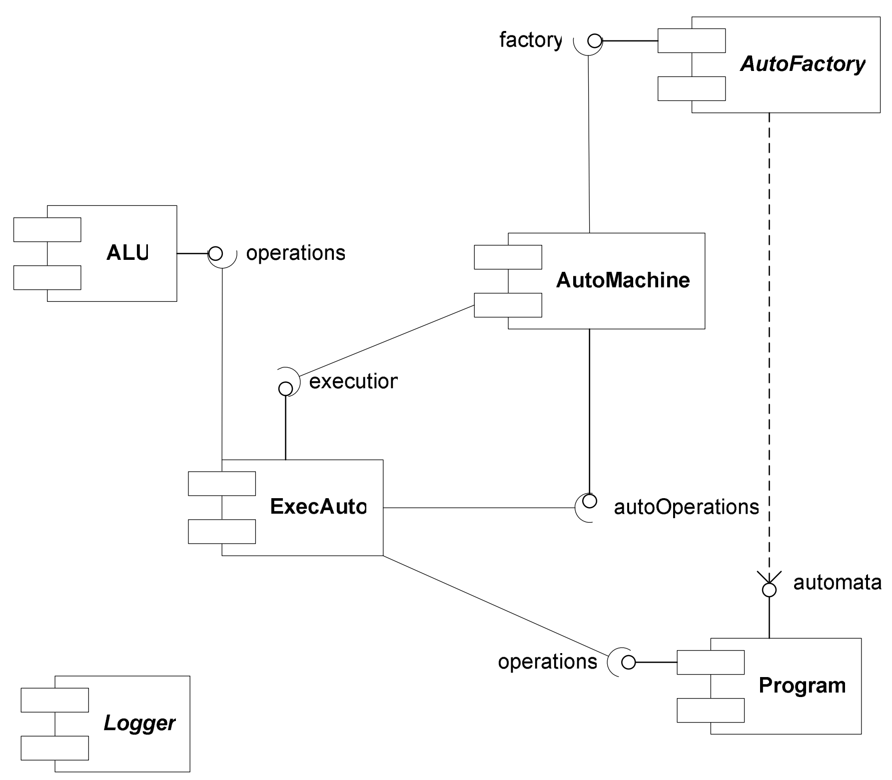


Рис. 4.1.1.

Компонентная структура автоматной машины

Данная структура компонентов является реализацией спроектированной ранее архитектуры (рис. 3.1.3.2).

Автоматная машина осуществляет непосредственно интерпретацию автоматной программы на основе представленного ранее алгоритма (рис. 3.1.3.1). Этот компонент интерпретирует экземпляры автоматов, предоставляемые ему фабрикой автоматов (*AutoFactory*), вызывая проверку условий и выполнение эффектов в блоке выполнения (ExecAuto).

Блок выполнения делегирует вычисление условий и выполнение операций интерпретируемой программе (Program) или использует встроенные арифметико-логические функции (ALU).

Служба ведения лога используется компонентами AutoMachine, ExecAuto, *AutoFactory* для записи основных действий и возникших ошибок. Таким образом осуществляется отслеживание работы машины автоматного программирования и отладка автоматных программ.

Фабрика автоматов создает экземпляры автоматов, используемых в интерпретируемой автоматной программе, связывает их с операциями для вычисления условий на переходах и выполнения эффектов. Кроме того, фабрика автоматов отвечает за хранение автоматной программы во время ее интерпретации и сборку мусора после окончания работы машины автоматного программирования.

Фабрика автоматов реализована двумя способами: в виде транслятора автоматной программы и с помощью применения метода раскрутки к машине автоматного программирования.

Транслятор автоматной программы создает автоматы на основе диаграмм состояний, построенных в Microsoft Visio, и связывает их с функциями динамически подключаемой библиотеки (dynamic link library) (рис. 4.1.2).

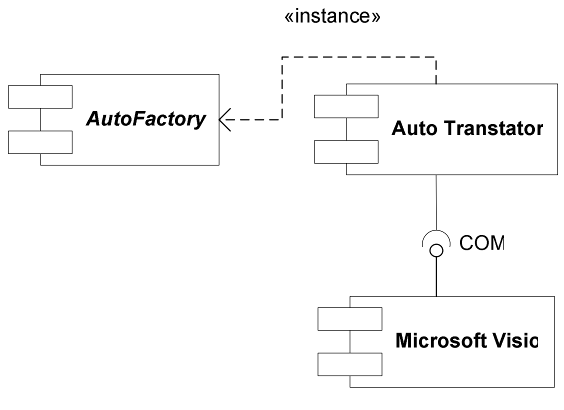


Рис. 4.1.2.

Транслятор автоматной программы

Семантика автоматной программы определена в виде автоматной программы (рис. 3.1.3.1). Следовательно, можно осуществить метод раскрутки и реализовать машину автоматного программирования в виде автоматной программы, интерпретируемой машиной автоматного программирования. Этот подход применяется следующим образом.

Компонент раскрутки BootStrap (рис. 4.1.3) реализует машину автоматного программирования в виде автоматной программы:

- компонент раскрутки является фабрикой единственного экземпляра автомата – автомата интерпретации автоматной программы (и как фабрика предоставляет интерфейс factory);
- компонент раскрутки является автоматной программой (и как автоматная программа предоставляет интерфейс operations);

* компонент раскрутки является автоматной машиной (и как автоматная машина использует интерфейсы execution и factory).

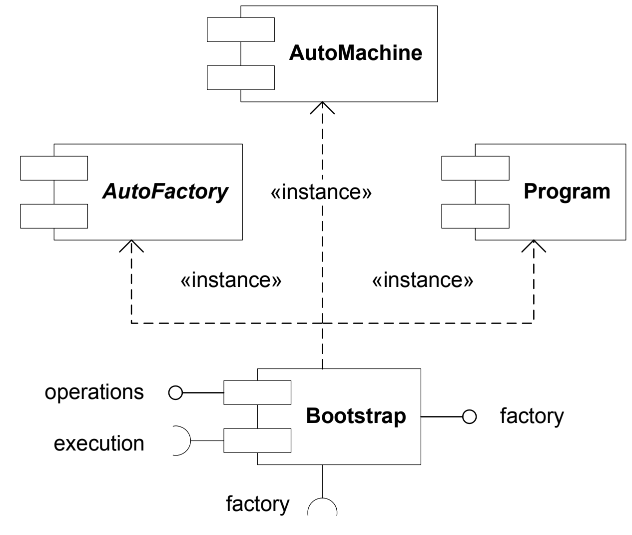


Рис. 4.1.3.

«Раскрученная» машина автоматного программирования

Интерфейс factory, предоставляемый компонентом раскрутки, используется машиной автоматного программирования для интерпретации создаваемого им автомата. Интерфейс factory, требуемый компонентом раскрутки, используется им для получения автоматов, которые он интерпретирует. С помощью интерфейса execution компонент раскрутки вычисляет условия и выполняет эффекты в интерпретируемых им автоматах. Предоставляемый компонентом раскрутки интерфейс operations используется блоком выполнения для его интерпретации. При этом фактически используется один блок выполнения (рис. 4.1.4).

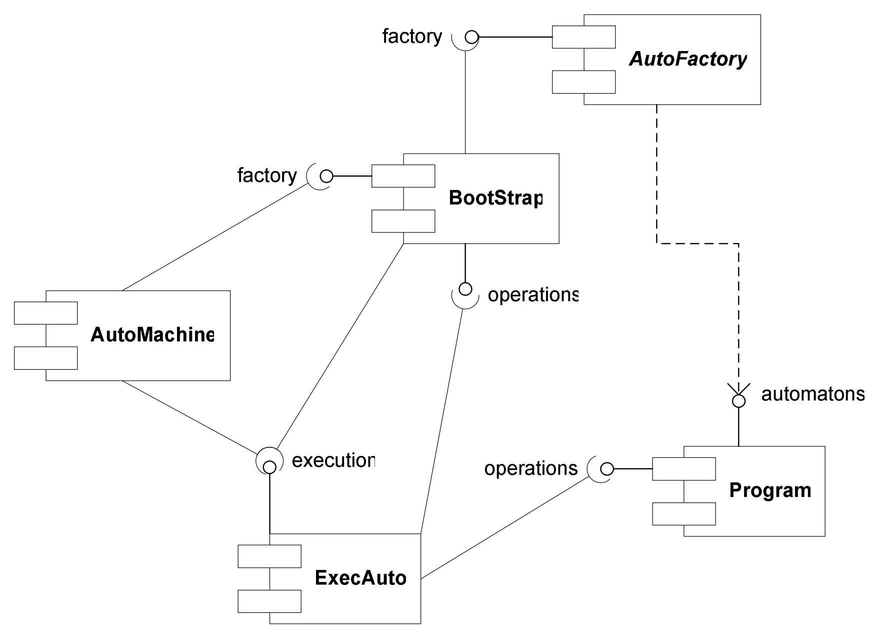


Рис. 4.1.4.

Применение метода раскрутки для машины автоматного программирования

4.2. Транслятор автоматной программы

Транслятор автоматной программы является одной из возможных реализаций фабрики автоматов. Он транслирует автоматную программу, представленную в виде двух файлов: *Visio*-файла с диаграммами состояний автоматов и файла динамически подключаемой библиотеки (рис. 4.2.1).

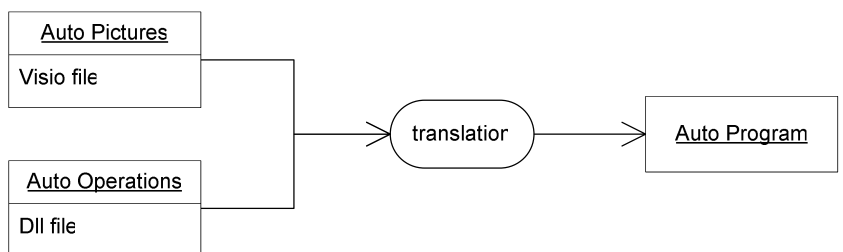


Рис. 4.2.1.

Трансляция автоматной программы на основе диаграмм состояний Microsoft Visio

Динамически подключаемая библиотека является реализацией объекта управления. Она содержит все операции, используемые автоматной программой для вычисления условий и выполнения эффектов. Следовательно, в процессе интерпретации автоматной программы при вычислении условий на переходах и выполнении эффектов в состояниях и на переходах фактически вызываются функции библиотеки.

Непосредственно автоматная программа описывается в виде диаграмм состояний, построенных с помощью *Microsoft Visio*. Для ее трансляции используется *COM*-интерфейс, предоставляемый этим редактором. Идея использования *COM*-интерфейса *MS Visio* позаимствована в работе [23].

В результате анализа *COM*-интерфейса *MS Visio* как графического синтаксиса автоматного программирования получена следующая модель используемого представления автоматной программы(рис. 4.2.2).

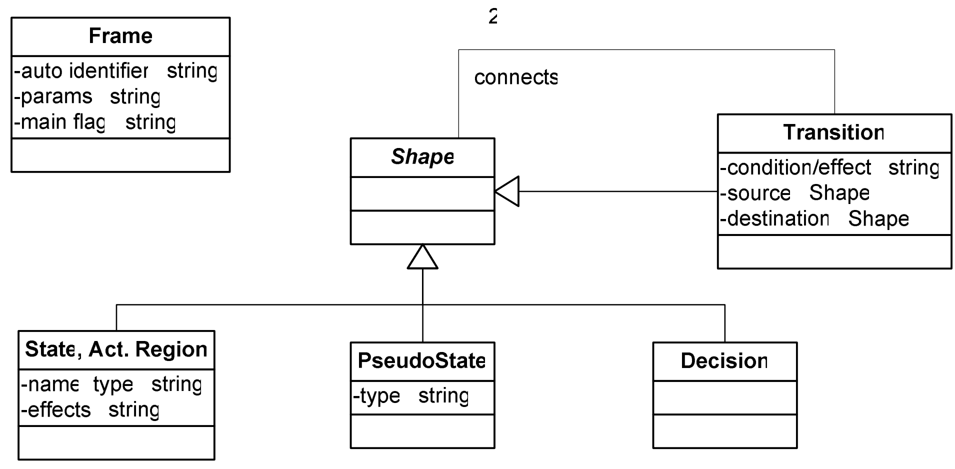


Рис. 4.2.2.

Модель синтаксиса диаграмм состояний, используемого в COM-интерфейсе MS Visio

Таким образом, автоматная программа, описанная в виде диаграмм состояний *MS Visio*, является экземпляром модели, представленной на рис. 4.2.2. Обозначим этот экземпляр автоматной программы *VA*. Для интерпретации машиной автоматного программирования необходим экземпляр метамодели, представленной на рис. 3.1.1.2 (обозначим его *IA*). Соответственно транслятор осуществляет преобразование полученной автоматной программы *VA* в целевую пригодную для интерпретации автоматную программу *IA*.

Заметим, что обе эти модели по существу отличаются друг от друга только способом представления графа автомата: в модели *VA* списки смежности определены для ребер графа (source: shape и destination: shape), в модели *IA* списки смежности определены для вершин графа (out: transitions). Поэтому для осуществления трансляции строится промежуточное представление графа автомата, объединяющее в себе свойства обоих моделей, а затем оно проецируется на целевую модель *IA*.

Кроме того, трансляция включает реализацию семантик вспомогательных элементов графического синтаксиса, представленных на рис. 3.1.2.1 и 3.1.2.2, а именно: исключение символов ветвления и преобразование точек входа и выхода, принадлежащих составному состоянию, в соответствующие условия и эффекты на переходах этого состояния. Эти преобразования также выполняются с помощью промежуточного представления и установленных для него связей.

На рисунке 4.2.3 приведен алгоритм построения промежуточного представления.

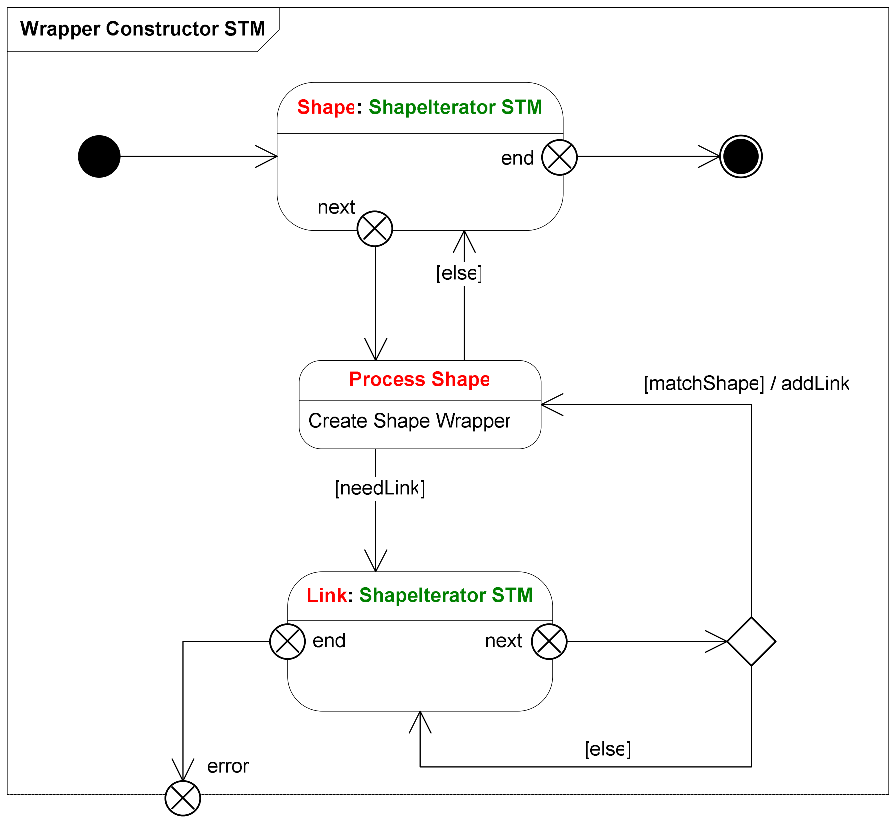


Рис. 4.2.3.

Алгоритм построения промежуточного представления автоматной программы

Для каждой фигуры Shape автоматной программы *VA* создается ее промежуточное представление (Shape Wrapper) и устанавливаются дополнительные связи со всеми ссылающимися на нее фигурами *VA*. Очевидно, что сложность этого построения – квадратичная от общего числа фигур.

Приведенный на рис. 4.2.3 автомат является частью автоматной программы, реализующей рассматриваемый транслятор. Таким образом, для реализации транслятора автоматной программы может быть также применен метод раскрутки.

4.3. Реализация компонентов

Все компоненты машины автоматного программирования реализованы на языке программирования *Object Pascal* в среде *Delphi,* что диктуется требованием совместимости с существующей реализацией системы *ЭРА* и ее предметного наполнения. На рис. 4.3.1 приведена структура кода разработанной программы: программные модули и их связи друг с другом.

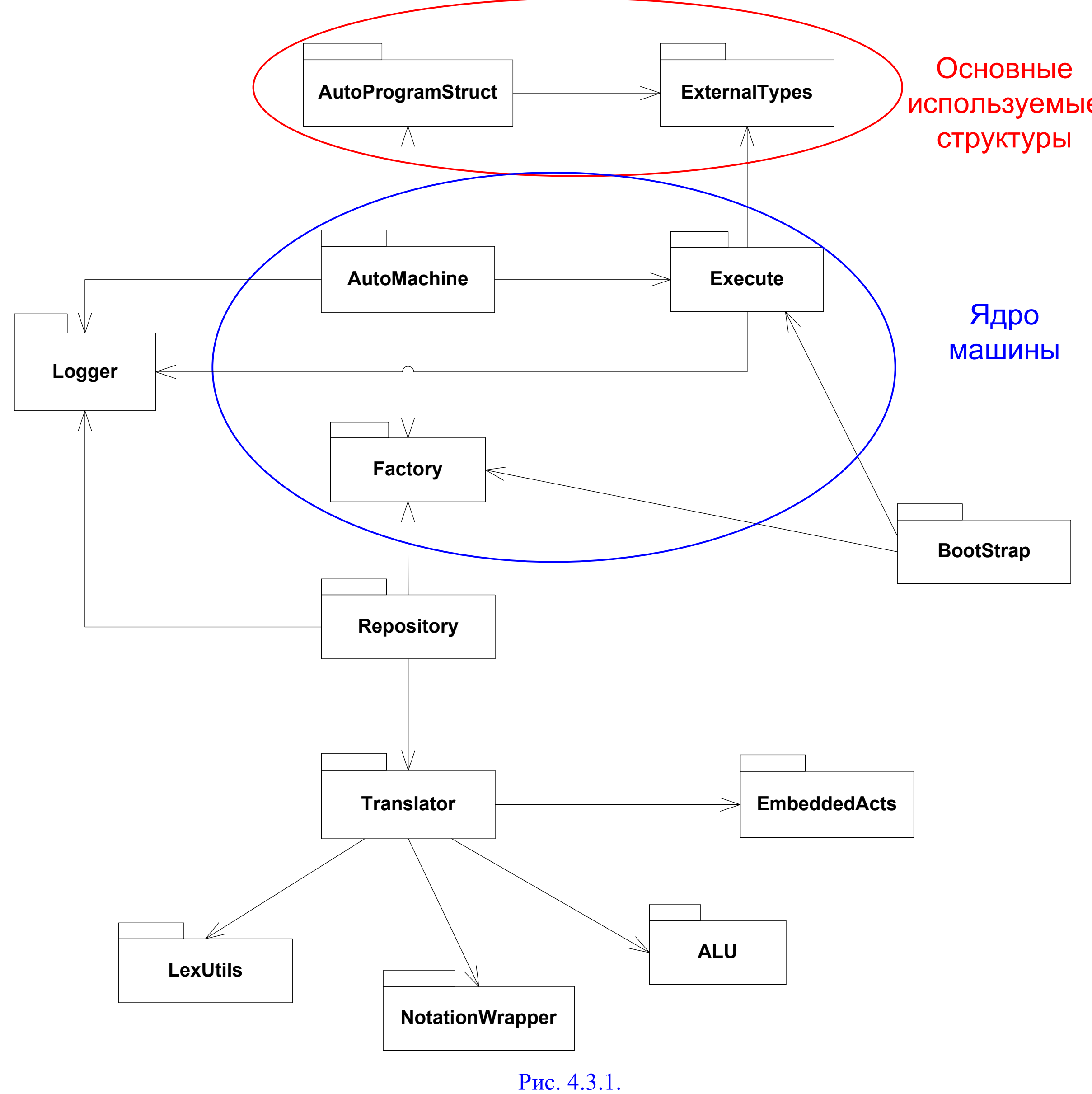


Рис. 4.3.1.

Структура кода разработанной программы

Более подробно эти модули описаны в табл.2.

Таблица 2. Описание программных модулей

Модуль	Содержание	Описание
`AutoProgramStruct`	Метамодель автоматной машины	Соответствует рис. 3.1.1.2.
`ExternalTypes`	Интерфейсы условий и эффектов	Инкапсулируют вызовы операций объекта управления
`Execute`	Блок выполнения	
`AutoMachine`	Интерпретатор автоматных программ	Соответствует рис. 3.1.3.1.
`Factory`	Интерфейс фабрики автоматов	Интерфейс factory

Logger	Служба ведения лога	Реализованы три уровня степени детализации лога
EmbeddedActs	Встроенная реализация интерфейсов условий и эффектов	Предопределенные способы инкапсуляции функций динамически подключаемой библиотеки
BootStrap	Компонент раскрутки	Является фабрикой, автоматной машиной и предоставляет функции, вызываемые во время интерпретации
Repository	Хранилище автоматов	Реализует интерфейс factory, создание автоматов делегирует транслятору
Translator	Транслятор автоматных программ	Вызывает *Visio*-приложение и использует его *COM*-интерфейс
NotationWrapper	Промежуточное представление автоматной программы	Хранит все ссылки, используемые для трансляции
LexUtils	Разбор строк	Использует библиотеки *Delphi* для работы со строками
ALU	Встроенные арифметико-логические функции	Целочисленные сложение, вычитание, сравнение с константой и т.д.

Машина автоматного программирования реализована в виде консольного приложения, командная строка (табл.3) которого имеет следующий формат:

```
AutoExecute.exe [-l<logging level>] [BOOTSTRAP] <program file name>
[<library file name>]
```

Таблица 3. Аргументы командной строки

Аргумент	**Описание**	**Возможные значения**	**Значение по умолчанию**
logging level	Уровень степени детализации ведения лога	-l(0|1|2)	-l1
BOOTSTRAP	Включение режима раскрутки	BOOTSTRAP	Выключен
program file	*Visio*-файл с описанием автоматов	*.vsd	Обязательный

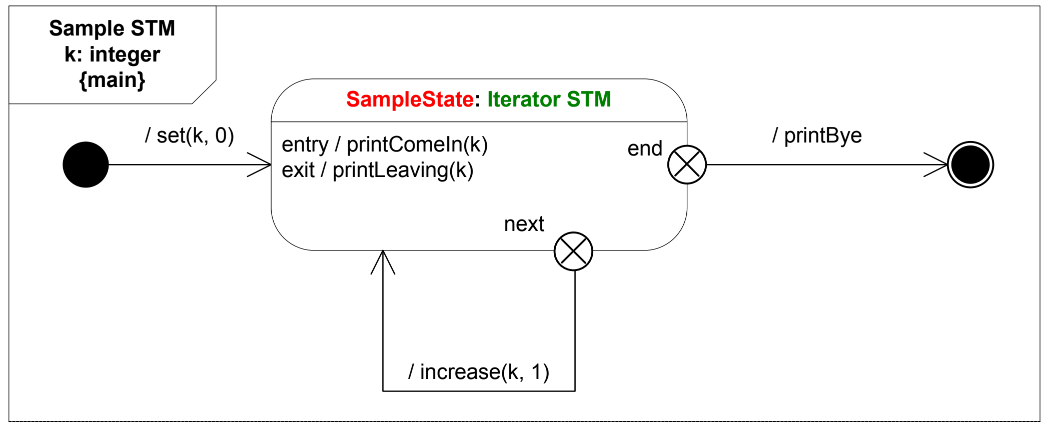
name	программы		аргумент
library file name	Файл динамически подключаемой библиотеки	*.dll	Необязательный аргумент

Таким образом, машина автоматного программирования может работать в двух режимах: без использования компонента Bootstrap или с реализацией раскрутки. Если динамически подключаемая библиотека отсутствует, то автоматная программа должна ссылаться только на встроенные операции машины автоматного программирования (арифметико-логические функции и операции управления машиной).

4.4. Тестирование

Основным результатом, подтверждающим корректность разработанной программы, является реализованный с ее помощью метод раскрутки.

На рис. 4.4.1 приведена автоматная программа, с помощью которой реализовано тестирование машины автоматного программирования.



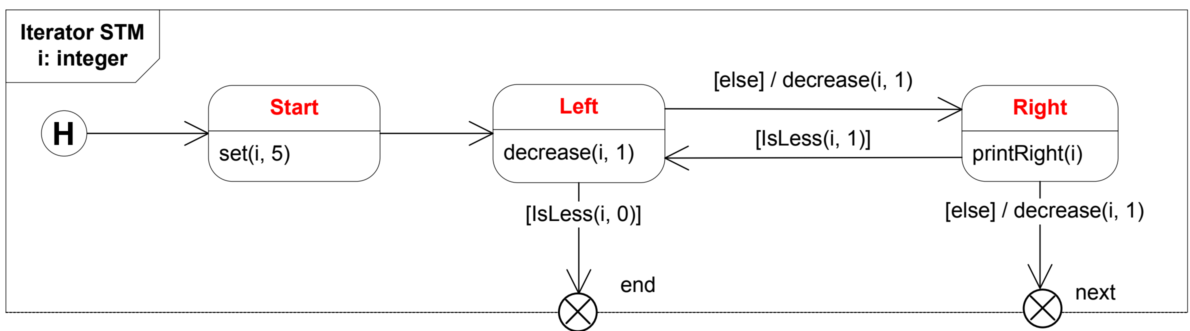


Рис. 4.4.1.

Автоматная программа, используемая для тестирования машины автоматного программирования

Эта автоматная программа использует арифметико-логические функции, целочисленные параметры и осуществляет печать их промежуточных значений. Интерпретация этой программы выполнена в двух режимах: с применением компонента

раскрутки и без него. В приложениях 1 и 2 приведены соответствующие логи работы машины автоматного программирования. Из их анализа и из полученного в результате интерпретации тестовой автоматной программы текстового выхода (рис. 4.4.2) можно сделать вывод, что метод раскрутки реализован, а машина автоматного программирования корректно работает для рассмотренных примеров.

```
Come in sample state:
parameter k = 0
in right:
parameter i = 3
Leaving sample state:
parameter k = 0
Come in sample state:
parameter k = 1
in right:
parameter i = 2
Leaving sample state:
parameter k = 1
Come in sample state:
parameter k = 2
in right:
parameter i = 1
Leaving sample state:
parameter k = 2
The end!
```

Рис. 4.4.2.

Результат интерпретации тестовой автоматной программы

Ниже приведены результаты тестирования (табл. 4) производительности машины автоматного программирования. Они получены для той же автоматной программы с помощью задания различных начальных значений для параметра i автомата Iterator STM.

Таблица 4. Результаты тестирования производительности

Начальное значение параметра i	**Время работы автоматной машины, миллисекунды**	**Режим раскрутки**	**Примерное число выполненных переходов**
1000	1265	+	11000
5000	5656	+	55000
10000	11360	+	110000
20000	23547	+	220000
1000	906	–	2000
5000	5062	–	10000
10000	10156	–	20000
20000	20938	–	40000

Примерное число выполненных переходов получено в результате анализа автоматной программы и алгоритма ее интерпретации. На данный момент не понятно,

почему время работы «раскрученной» машины не намного больше аналогичного времени «нераскрученной» машины, хотя количество переходов в «раскрученной» в несколько раз больше.

Заключение

В результате проведенной работы спроектирован и программно реализован метод определения проблемно-ориентированных языков интерпретируемыми автоматами. В том числе выполнены следующие задачи:

- рассмотрены существующие в настоящее время подходы к определению проблемно-ориентированных языков;
- проведен анализ диаграмм классов *UML* как средства задания проблемно-ориентированного языка;
- определен полуавтоматический метод сведения фрагментов метамодели языка во фрагменты распознающих автоматов;
- спроектированы автоматы, описывающие семантику табличных выражений проблемно-ориентированного языка *СЛОН*;
- спроектирована и реализована машина автоматного программирования;
- к машине автоматного программирования применен метод раскрутки.

С помощью рассмотренного метода удалось описать наиболее сложный с точки зрения семантики фрагмент языка *СЛОН*: табличные выражения. Теперь с помощью машины автоматного программирования можно получить языковый процессор этого фрагмента. В дальнейшем планируются следующие направления работы:

- реализация всего проблемно-ориентированного языка *СЛОН*;
- расширение машины автоматного программирования событийной моделью (добавление устойчивых состояний);
- добавление в машину автоматного программирования поддержки параллельных процессов;
- разработка редактора автоматных программ;
- реализация новой версии языка *СЛОН*.

Литература

1. *Fowler M.* «Language Workbenches: The Killer-App for Domain Specific Languages?» www.martinfowler.com/articles 12 June 2005
2. *Dmitriev S.* «Language Oriented Programming: The Next Programming Paradigm» www.onboard.jetbrains.com February 2005
3. Оллонгрен А. Определение языков программирования интерпретирующими автоматами. М.: Мир, 1977.
4. Сайт по автоматному программированию и мотивации к творчеству СПбГУИТМО. Кафедра «Технологии программирования». http://is.ifmo.ru/
4. *Krasinsky G. A., Novikov F. A., Skripnichenko V. I.* Problem Oriented Language for Ephemeris Astronomy and its Realization in System ERA. Cel. Mech., 1989. Vol. 45, pp. 219–229.
5. *Новиков Ф. А.* Архитектура системы ЭРА – табличный подход к обработке данных. Л.: ИПА РАН, 1990.
6. *Лавров С. С.* Программирование. Математические основы, средства, теория. БХВ-Петербург, 2001.
7. *Ахо А., Сети Р., Ульман Д.* Компиляторы: принципы, технологии и инструменты. Вильямс, 2003.
8. *The Vienna Development Method.* 2007. www.vdmportal.org,
9. *JetBrains*: Meta Programming System. 2005. http://www.jetbrains.com/mps/
10. *Fowler M.* A Language Workbench in Action – MPS. 2005. http://martinfowler.com/articles/mpsAgree.html, 2005
11. *MetaEdit+ Domain Specific Modeling tools.* http://www.metacase.com/products.html, 1995 – 2008.
12. *Kelly S., Tolvanen J-P.* Domain-Specific Modeling: Enabling Full Code Generation. Wiley-IEEE Computer Society Press, 2008.
13. *Eclipse:* Generative Modeling Technologies (GMT) project. 2002–2008. http://www.eclipse.org/gmt/
14. *Intentional Software. 2002–2008.* http://www.intentsoft.com/
15. *Software Factories.* Industrialized Software Development. 2004. http://www.softwarefactories.com
16. *Крашенинников С. В., Кривоногов А. В., Назаров А. А., Новиков Ф. А., Скрипниченко В. И.* Система таблично ориентированного программирования: 32-разрядная версия. СПб.: ИПА РАН, 1999.
17. *OMG* «Unified Modeling Language: Superstructure». Version 2.1.1. 2007.

18. *Новиков Ф. , Яценко А.* Microsoft Office в целом. СПб БХВ, 2003.
19. *Буч Г., Рамбо Д., Джекобсон А.* Язык UML. Руководство пользователя. М.: ДМК-пресс, 2007.
20. *Новиков Ф. А.* Анализ и проектирование на UML. Материалы курса. 2006. http://ru.sun.com/research/teachingmaterials.html
21. *Гуров В. С., Мазин М. А., Шалыто А.А.* Автоматическое завершение ввода условий в диаграммах состояний. 2008. http://is.ifmo.ru/works/_2008-02-28_auto_stop.pdf
22. *UniMod. Executable UML*. 2003–2005. http://unimod.sourceforge.net/
23. *Канжелев С. Ю., Шалыто А. А.* Преобразование графов переходов, представленных в формате MS Visio, в исходные коды программ для различных языков программирования (инструментальное средство MetaAuto). Проектная документация. 2005. http://is.ifmo.ru/
24. *Евдокимов В. И., Козаченко В. И., Нейман Л. А., Румянцев В. В.* Охрана труда и окружающей среды Учебное пособие. СПб.: СПГААП, 1993.
25. *СанПиН 2.2.2/2.4.1340-03.* Гигиенические требования к персональным электронно-вычислительным машинам и организации работы. СПб.: СПбГПУ, 2004.

Приложение 1. Лог выполнения автоматной программы

```
CREATE AUTO OF Sample STM-TYPE (id = 0)
GO TO AUTO (id = 0)
STATE: Initial State
PARAMETERS OF AUTOMATON (id = 0):
   parameter k = 0
CHOSE TRANSITION FROM STATE: Initial State
UNMOTIVATED TRANSITION => TRUE
ON TRANSITION
PARAMETERS OF AUTOMATON (id = 0):
   parameter k = 0
EFFECT
   ACTION set
STATE: SampleState
PARAMETERS OF AUTOMATON (id = 0):
   parameter k = 0
EFFECT
   ACTION printComeIn
CREATE AUTO OF Iterator STM-TYPE (id = 1)
GO FROM AUTO (id = 0) TO AUTO (id = 1)
STATE: Shallow History
PARAMETERS OF AUTOMATON (id = 1):
   parameter i = 0
CHOSE TRANSITION FROM STATE: Shallow History
UNMOTIVATED TRANSITION => TRUE
ON TRANSITION
PARAMETERS OF AUTOMATON (id = 1):
   parameter i = 0
STATE: Start
PARAMETERS OF AUTOMATON (id = 1):
   parameter i = 0
EFFECT
   ACTION set
LEAVING STATE: Start
CHOSE TRANSITION FROM STATE: Start
UNMOTIVATED TRANSITION => TRUE
ON TRANSITION
PARAMETERS OF AUTOMATON (id = 1):
   parameter i = 5
STATE: Left
PARAMETERS OF AUTOMATON (id = 1):
   parameter i = 5
EFFECT
   ACTION decrease
LEAVING STATE: Left
CHOSE TRANSITION FROM STATE: Left
COMPUTED CONDITION: IsLess = FALSE
UNMOTIVATED TRANSITION => TRUE
ON TRANSITION
PARAMETERS OF AUTOMATON (id = 1):
   parameter i = 4
EFFECT
   ACTION decrease
STATE: Right
PARAMETERS OF AUTOMATON (id = 1):
   parameter i = 3
EFFECT
   ACTION printRight
LEAVING STATE: Right
CHOSE TRANSITION FROM STATE: Right
COMPUTED CONDITION: IsLess = FALSE
UNMOTIVATED TRANSITION => TRUE
ON TRANSITION
```

```
PARAMETERS OF AUTOMATON (id = 1):
   parameter i = 3
EFFECT
   ACTION decrease
GO FROM AUTO (id = 1) TO AUTO (id = 0)
LEAVING STATE: SampleState
EFFECT
   ACTION printLeaving
CHOSE TRANSITION FROM STATE: SampleState
COMPUTED CONDITION: exit point = next = TRUE
ON TRANSITION
PARAMETERS OF AUTOMATON (id = 0):
   parameter k = 0
EFFECT
   ACTION increase
STATE: SampleState
PARAMETERS OF AUTOMATON (id = 0):
   parameter k = 1
EFFECT
   ACTION printComeIn
GO FROM AUTO (id = 0) TO AUTO (id = 1)
STATE: Shallow History
PARAMETERS OF AUTOMATON (id = 1):
   parameter i = 2
STATE: Right
PARAMETERS OF AUTOMATON (id = 1):
   parameter i = 2
EFFECT
   ACTION printRight
LEAVING STATE: Right
CHOSE TRANSITION FROM STATE: Right
COMPUTED CONDITION: IsLess = FALSE
UNMOTIVATED TRANSITION => TRUE
ON TRANSITION
PARAMETERS OF AUTOMATON (id = 1):
   parameter i = 2
EFFECT
   ACTION decrease
GO FROM AUTO (id = 1) TO AUTO (id = 0)
LEAVING STATE: SampleState
EFFECT
   ACTION printLeaving
CHOSE TRANSITION FROM STATE: SampleState
COMPUTED CONDITION: exit point = next = TRUE
ON TRANSITION
PARAMETERS OF AUTOMATON (id = 0):
   parameter k = 1
EFFECT
   ACTION increase
STATE: SampleState
PARAMETERS OF AUTOMATON (id = 0):
   parameter k = 2
EFFECT
   ACTION printComeIn
GO FROM AUTO (id = 0) TO AUTO (id = 1)
STATE: Shallow History
PARAMETERS OF AUTOMATON (id = 1):
   parameter i = 1
STATE: Right
PARAMETERS OF AUTOMATON (id = 1):
   parameter i = 1
EFFECT
   ACTION printRight
LEAVING STATE: Right
```

```
CHOSE TRANSITION FROM STATE: Right
COMPUTED CONDITION: IsLess = TRUE
ON TRANSITION
PARAMETERS OF AUTOMATON (id = 1):
   parameter i = 1
STATE: Left
PARAMETERS OF AUTOMATON (id = 1):
   parameter i = 1
EFFECT
   ACTION decrease
LEAVING STATE: Left
CHOSE TRANSITION FROM STATE: Left
COMPUTED CONDITION: IsLess = TRUE
ON TRANSITION
PARAMETERS OF AUTOMATON (id = 1):
   parameter i = 0
GO FROM AUTO (id = 1) TO AUTO (id = 0)
LEAVING STATE: SampleState
EFFECT
   ACTION printLeaving
CHOSE TRANSITION FROM STATE: SampleState
COMPUTED CONDITION: exit point = next = FALSE
COMPUTED CONDITION: exit point = end = TRUE
ON TRANSITION
PARAMETERS OF AUTOMATON (id = 0):
   parameter k = 2
EFFECT
   ACTION printBye
GO FROM AUTO (id = 0) TO NOWHERE
TERMINATE!
```

Приложение 2. Лог выполнения автоматной программы в режиме раскрутки

```
CREATE AUTO OF Automaton of auto machine (bootstrapped) -TYPE (id = 0)
GO TO AUTO (id = 0)
STATE: Bootstrapped AM: Initial

CHOSE TRANSITION FROM STATE: Bootstrapped AM: Initial
UNMOTIVATED TRANSITION => TRUE
ON TRANSITION

EFFECT
   ACTION Bootstrapped AM: Init machine
CREATE AUTO OF Sample STM-TYPE (id = 0)
   ACTION Bootstrapped AM: curState := start
STATE: Bootstrapped AM: Current state

LEAVING STATE: Bootstrapped AM: Current state
CHOSE TRANSITION FROM STATE: Bootstrapped AM: Current state
COMPUTED CONDITION: Bootstrapped AM: curState is composite = FALSE
COMPUTED CONDITION: Bootstrapped AM: curState is simple = FALSE
COMPUTED CONDITION: Bootstrapped AM: (curState is pseudo) AND (curState is history) AND NOT (history is null) = FALSE
COMPUTED CONDITION: Bootstrapped AM: curState is pseudo = TRUE
ON TRANSITION

EFFECT
   ACTION Bootstrapped AM: curTrans := out.first
STATE: Bootstrapped AM: Choose transition

LEAVING STATE: Bootstrapped AM: Choose transition
CHOSE TRANSITION FROM STATE: Bootstrapped AM: Choose transition
COMPUTED CONDITION: Bootstrapped AM: (curTrans is null) AND (start state is history) = FALSE
COMPUTED CONDITION: Bootstrapped AM: curTrans is null = FALSE
UNMOTIVATED TRANSITION => TRUE
COMPUTED CONDITION: Bootstrapped AM: compute(curTrans.condition) = TRUE
ON TRANSITION

STATE: Bootstrapped AM: On transition

EFFECT
   ACTION Bootstrapped AM: DoEffect(curTrans.effect)
EFFECT
   ACTION set
LEAVING STATE: Bootstrapped AM: On transition
CHOSE TRANSITION FROM STATE: Bootstrapped AM: On transition
COMPUTED CONDITION: Bootstrapped AM: (curTrans.dest is final) AND (curTrans.dest is exitPoint) AND (start state is history) = FALSE
COMPUTED CONDITION: Bootstrapped AM: (curTrans.dest is final) AND (curTrans.dest is exitPoint) = FALSE
COMPUTED CONDITION: Bootstrapped AM: (curTrans.dest is final) AND (start state is history) = FALSE
COMPUTED CONDITION: Bootstrapped AM: curTrans.dest is final = FALSE
UNMOTIVATED TRANSITION => TRUE
ON TRANSITION

EFFECT
   ACTION Bootstrapped AM: curState := curTrans.dest
STATE: Bootstrapped AM: Current state

LEAVING STATE: Bootstrapped AM: Current state
CHOSE TRANSITION FROM STATE: Bootstrapped AM: Current state
```

```
COMPUTED CONDITION: Bootstrapped AM: curState is composite = TRUE
ON TRANSITION

STATE: Bootstrapped AM: GoTo

EFFECT
   ACTION Bootstrapped AM: DoEffect(curState.entry)
EFFECT
   ACTION printComeIn
   ACTION Bootstrapped AM: currAuto := curState.insAuto
CREATE AUTO OF Iterator STM-TYPE (id = 1)
   ACTION Bootstrapped AM: curState := find(owner.entryPoint)
LEAVING STATE: Bootstrapped AM: GoTo
CHOSE TRANSITION FROM STATE: Bootstrapped AM: GoTo
COMPUTED CONDITION: Bootstrapped AM: curState is null = TRUE
ON TRANSITION

EFFECT
   ACTION Bootstrapped AM: curState := start
STATE: Bootstrapped AM: Current state

LEAVING STATE: Bootstrapped AM: Current state
CHOSE TRANSITION FROM STATE: Bootstrapped AM: Current state
COMPUTED CONDITION: Bootstrapped AM: curState is composite = FALSE
COMPUTED CONDITION: Bootstrapped AM: curState is simple = FALSE
COMPUTED CONDITION: Bootstrapped AM: (curState is pseudo) AND (curState is
history) AND NOT (history is null) = FALSE
COMPUTED CONDITION: Bootstrapped AM: curState is pseudo = TRUE
ON TRANSITION

EFFECT
   ACTION Bootstrapped AM: curTrans := out.first
STATE: Bootstrapped AM: Choose transition

LEAVING STATE: Bootstrapped AM: Choose transition
CHOSE TRANSITION FROM STATE: Bootstrapped AM: Choose transition
COMPUTED CONDITION: Bootstrapped AM: (curTrans is null) AND (start state is
history) = FALSE
COMPUTED CONDITION: Bootstrapped AM: curTrans is null = FALSE
UNMOTIVATED TRANSITION => TRUE
COMPUTED CONDITION: Bootstrapped AM: compute(curTrans.condition) = TRUE
ON TRANSITION

STATE: Bootstrapped AM: On transition

EFFECT
   ACTION Bootstrapped AM: DoEffect(curTrans.effect)
LEAVING STATE: Bootstrapped AM: On transition
CHOSE TRANSITION FROM STATE: Bootstrapped AM: On transition
COMPUTED CONDITION: Bootstrapped AM: (curTrans.dest is final) AND
(curTrans.dest is exitPoint) AND (start state is history) = FALSE
COMPUTED CONDITION: Bootstrapped AM: (curTrans.dest is final) AND
(curTrans.dest is exitPoint) = FALSE
COMPUTED CONDITION: Bootstrapped AM: (curTrans.dest is final) AND (start
state is history) = FALSE
COMPUTED CONDITION: Bootstrapped AM: curTrans.dest is final = FALSE
UNMOTIVATED TRANSITION => TRUE
ON TRANSITION

EFFECT
   ACTION Bootstrapped AM: curState := curTrans.dest
STATE: Bootstrapped AM: Current state

LEAVING STATE: Bootstrapped AM: Current state
```

```
CHOSE TRANSITION FROM STATE: Bootstrapped AM: Current state
COMPUTED CONDITION: Bootstrapped AM: curState is composite = FALSE
COMPUTED CONDITION: Bootstrapped AM: curState is simple = TRUE
ON TRANSITION

EFFECT
   ACTION Bootstrapped AM: DoEffect(curState.entry)
EFFECT
   ACTION set
STATE: Bootstrapped AM: Leaving current state

EFFECT
   ACTION Bootstrapped AM: DoEffect(curState.exit)
LEAVING STATE: Bootstrapped AM: Leaving current state
CHOSE TRANSITION FROM STATE: Bootstrapped AM: Leaving current state
UNMOTIVATED TRANSITION => TRUE
ON TRANSITION

EFFECT
   ACTION Bootstrapped AM: curTrans := out.first
STATE: Bootstrapped AM: Choose transition

LEAVING STATE: Bootstrapped AM: Choose transition
CHOSE TRANSITION FROM STATE: Bootstrapped AM: Choose transition
COMPUTED CONDITION: Bootstrapped AM: (curTrans is null) AND (start state is
history) = FALSE
COMPUTED CONDITION: Bootstrapped AM: curTrans is null = FALSE
UNMOTIVATED TRANSITION => TRUE
COMPUTED CONDITION: Bootstrapped AM: compute(curTrans.condition) = TRUE
ON TRANSITION

STATE: Bootstrapped AM: On transition

EFFECT
   ACTION Bootstrapped AM: DoEffect(curTrans.effect)
LEAVING STATE: Bootstrapped AM: On transition
CHOSE TRANSITION FROM STATE: Bootstrapped AM: On transition
COMPUTED CONDITION: Bootstrapped AM: (curTrans.dest is final) AND
(curTrans.dest is exitPoint) AND (start state is history) = FALSE
COMPUTED CONDITION: Bootstrapped AM: (curTrans.dest is final) AND
(curTrans.dest is exitPoint) = FALSE
COMPUTED CONDITION: Bootstrapped AM: (curTrans.dest is final) AND (start
state is history) = FALSE
COMPUTED CONDITION: Bootstrapped AM: curTrans.dest is final = FALSE
UNMOTIVATED TRANSITION => TRUE
ON TRANSITION

EFFECT
   ACTION Bootstrapped AM: curState := curTrans.dest
STATE: Bootstrapped AM: Current state

LEAVING STATE: Bootstrapped AM: Current state
CHOSE TRANSITION FROM STATE: Bootstrapped AM: Current state
COMPUTED CONDITION: Bootstrapped AM: curState is composite = FALSE
COMPUTED CONDITION: Bootstrapped AM: curState is simple = TRUE
ON TRANSITION

EFFECT
   ACTION Bootstrapped AM: DoEffect(curState.entry)
EFFECT
   ACTION decrease
STATE: Bootstrapped AM: Leaving current state

EFFECT
```

```
   ACTION Bootstrapped AM: DoEffect(curState.exit)
LEAVING STATE: Bootstrapped AM: Leaving current state
CHOSE TRANSITION FROM STATE: Bootstrapped AM: Leaving current state
UNMOTIVATED TRANSITION => TRUE
ON TRANSITION

EFFECT
   ACTION Bootstrapped AM: curTrans := out.first
STATE: Bootstrapped AM: Choose transition

LEAVING STATE: Bootstrapped AM: Choose transition
CHOSE TRANSITION FROM STATE: Bootstrapped AM: Choose transition
COMPUTED CONDITION: Bootstrapped AM: (curTrans is null) AND (start state is history) = FALSE
COMPUTED CONDITION: Bootstrapped AM: curTrans is null = FALSE
COMPUTED CONDITION: IsLess = FALSE
COMPUTED CONDITION: Bootstrapped AM: compute(curTrans.condition) = FALSE
UNMOTIVATED TRANSITION => TRUE
ON TRANSITION

EFFECT
   ACTION Bootstrapped AM: curTrans := out.next
STATE: Bootstrapped AM: Choose transition

LEAVING STATE: Bootstrapped AM: Choose transition
CHOSE TRANSITION FROM STATE: Bootstrapped AM: Choose transition
COMPUTED CONDITION: Bootstrapped AM: (curTrans is null) AND (start state is history) = FALSE
COMPUTED CONDITION: Bootstrapped AM: curTrans is null = FALSE
UNMOTIVATED TRANSITION => TRUE
COMPUTED CONDITION: Bootstrapped AM: compute(curTrans.condition) = TRUE
ON TRANSITION

STATE: Bootstrapped AM: On transition

EFFECT
   ACTION Bootstrapped AM: DoEffect(curTrans.effect)
EFFECT
   ACTION decrease
LEAVING STATE: Bootstrapped AM: On transition
CHOSE TRANSITION FROM STATE: Bootstrapped AM: On transition
COMPUTED CONDITION: Bootstrapped AM: (curTrans.dest is final) AND (curTrans.dest is exitPoint) AND (start state is history) = FALSE
COMPUTED CONDITION: Bootstrapped AM: (curTrans.dest is final) AND (curTrans.dest is exitPoint) = FALSE
COMPUTED CONDITION: Bootstrapped AM: (curTrans.dest is final) AND (start state is history) = FALSE
COMPUTED CONDITION: Bootstrapped AM: curTrans.dest is final = FALSE
UNMOTIVATED TRANSITION => TRUE
ON TRANSITION

EFFECT
   ACTION Bootstrapped AM: curState := curTrans.dest
STATE: Bootstrapped AM: Current state

LEAVING STATE: Bootstrapped AM: Current state
CHOSE TRANSITION FROM STATE: Bootstrapped AM: Current state
COMPUTED CONDITION: Bootstrapped AM: curState is composite = FALSE
COMPUTED CONDITION: Bootstrapped AM: curState is simple = TRUE
ON TRANSITION

EFFECT
   ACTION Bootstrapped AM: DoEffect(curState.entry)
EFFECT
```

```
   ACTION printRight
STATE: Bootstrapped AM: Leaving current state

EFFECT
   ACTION Bootstrapped AM: DoEffect(curState.exit)
LEAVING STATE: Bootstrapped AM: Leaving current state
CHOSE TRANSITION FROM STATE: Bootstrapped AM: Leaving current state
UNMOTIVATED TRANSITION => TRUE
ON TRANSITION

EFFECT
   ACTION Bootstrapped AM: curTrans := out.first
STATE: Bootstrapped AM: Choose transition

LEAVING STATE: Bootstrapped AM: Choose transition
CHOSE TRANSITION FROM STATE: Bootstrapped AM: Choose transition
COMPUTED CONDITION: Bootstrapped AM: (curTrans is null) AND (start state is
history) = FALSE
COMPUTED CONDITION: Bootstrapped AM: curTrans is null = FALSE
COMPUTED CONDITION: IsLess = FALSE
COMPUTED CONDITION: Bootstrapped AM: compute(curTrans.condition) = FALSE
UNMOTIVATED TRANSITION => TRUE
ON TRANSITION

EFFECT
   ACTION Bootstrapped AM: curTrans := out.next
STATE: Bootstrapped AM: Choose transition

LEAVING STATE: Bootstrapped AM: Choose transition
CHOSE TRANSITION FROM STATE: Bootstrapped AM: Choose transition
COMPUTED CONDITION: Bootstrapped AM: (curTrans is null) AND (start state is
history) = FALSE
COMPUTED CONDITION: Bootstrapped AM: curTrans is null = FALSE
UNMOTIVATED TRANSITION => TRUE
COMPUTED CONDITION: Bootstrapped AM: compute(curTrans.condition) = TRUE
ON TRANSITION

STATE: Bootstrapped AM: On transition

EFFECT
   ACTION Bootstrapped AM: DoEffect(curTrans.effect)
EFFECT
   ACTION decrease
LEAVING STATE: Bootstrapped AM: On transition
CHOSE TRANSITION FROM STATE: Bootstrapped AM: On transition
COMPUTED CONDITION: Bootstrapped AM: (curTrans.dest is final) AND
(curTrans.dest is exitPoint) AND (start state is history) = TRUE
ON TRANSITION

EFFECT
   ACTION Bootstrapped AM: exitPoint := curTrans.dest.name
   ACTION Bootstrapped AM: history := curState
STATE: Bootstrapped AM: The End

EFFECT
   ACTION Bootstrapped AM: currAuto := caller
LEAVING STATE: Bootstrapped AM: The End
CHOSE TRANSITION FROM STATE: Bootstrapped AM: The End
COMPUTED CONDITION: Bootstrapped AM: currAuto is null = FALSE
UNMOTIVATED TRANSITION => TRUE
ON TRANSITION

STATE: Bootstrapped AM: Leaving current state
```

```
EFFECT
   ACTION Bootstrapped AM: DoEffect(curState.exit)
EFFECT
   ACTION printLeaving
LEAVING STATE: Bootstrapped AM: Leaving current state
CHOSE TRANSITION FROM STATE: Bootstrapped AM: Leaving current state
UNMOTIVATED TRANSITION => TRUE
ON TRANSITION

EFFECT
   ACTION Bootstrapped AM: curTrans := out.first
STATE: Bootstrapped AM: Choose transition

LEAVING STATE: Bootstrapped AM: Choose transition
CHOSE TRANSITION FROM STATE: Bootstrapped AM: Choose transition
COMPUTED CONDITION: Bootstrapped AM: (curTrans is null) AND (start state is history) = FALSE
COMPUTED CONDITION: Bootstrapped AM: curTrans is null = FALSE
COMPUTED CONDITION: exit point = next = TRUE
COMPUTED CONDITION: Bootstrapped AM: compute(curTrans.condition) = TRUE
ON TRANSITION

STATE: Bootstrapped AM: On transition

EFFECT
   ACTION Bootstrapped AM: DoEffect(curTrans.effect)
EFFECT
   ACTION increase
LEAVING STATE: Bootstrapped AM: On transition
CHOSE TRANSITION FROM STATE: Bootstrapped AM: On transition
COMPUTED CONDITION: Bootstrapped AM: (curTrans.dest is final) AND (curTrans.dest is exitPoint) AND (start state is history) = FALSE
COMPUTED CONDITION: Bootstrapped AM: (curTrans.dest is final) AND (curTrans.dest is exitPoint) = FALSE
COMPUTED CONDITION: Bootstrapped AM: (curTrans.dest is final) AND (start state is history) = FALSE
COMPUTED CONDITION: Bootstrapped AM: curTrans.dest is final = FALSE
UNMOTIVATED TRANSITION => TRUE
ON TRANSITION

EFFECT
   ACTION Bootstrapped AM: curState := curTrans.dest
STATE: Bootstrapped AM: Current state

LEAVING STATE: Bootstrapped AM: Current state
CHOSE TRANSITION FROM STATE: Bootstrapped AM: Current state
COMPUTED CONDITION: Bootstrapped AM: curState is composite = TRUE
ON TRANSITION

STATE: Bootstrapped AM: GoTo

EFFECT
   ACTION Bootstrapped AM: DoEffect(curState.entry)
EFFECT
   ACTION printComeIn
   ACTION Bootstrapped AM: currAuto := curState.insAuto
   ACTION Bootstrapped AM: curState := find(owner.entryPoint)
LEAVING STATE: Bootstrapped AM: GoTo
CHOSE TRANSITION FROM STATE: Bootstrapped AM: GoTo
COMPUTED CONDITION: Bootstrapped AM: curState is null = TRUE
ON TRANSITION

EFFECT
   ACTION Bootstrapped AM: curState := start
```

```
STATE: Bootstrapped AM: Current state

LEAVING STATE: Bootstrapped AM: Current state
CHOSE TRANSITION FROM STATE: Bootstrapped AM: Current state
COMPUTED CONDITION: Bootstrapped AM: curState is composite = FALSE
COMPUTED CONDITION: Bootstrapped AM: curState is simple = FALSE
COMPUTED CONDITION: Bootstrapped AM: (curState is pseudo) AND (curState is
history) AND NOT (history is null) = TRUE
ON TRANSITION

EFFECT
   ACTION Bootstrapped AM: curState := history
STATE: Bootstrapped AM: Current state

LEAVING STATE: Bootstrapped AM: Current state
CHOSE TRANSITION FROM STATE: Bootstrapped AM: Current state
COMPUTED CONDITION: Bootstrapped AM: curState is composite = FALSE
COMPUTED CONDITION: Bootstrapped AM: curState is simple = TRUE
ON TRANSITION

EFFECT
   ACTION Bootstrapped AM: DoEffect(curState.entry)
EFFECT
   ACTION printRight
STATE: Bootstrapped AM: Leaving current state

EFFECT
   ACTION Bootstrapped AM: DoEffect(curState.exit)
LEAVING STATE: Bootstrapped AM: Leaving current state
CHOSE TRANSITION FROM STATE: Bootstrapped AM: Leaving current state
UNMOTIVATED TRANSITION => TRUE
ON TRANSITION

EFFECT
   ACTION Bootstrapped AM: curTrans := out.first
STATE: Bootstrapped AM: Choose transition

LEAVING STATE: Bootstrapped AM: Choose transition
CHOSE TRANSITION FROM STATE: Bootstrapped AM: Choose transition
COMPUTED CONDITION: Bootstrapped AM: (curTrans is null) AND (start state is
history) = FALSE
COMPUTED CONDITION: Bootstrapped AM: curTrans is null = FALSE
COMPUTED CONDITION: IsLess = FALSE
COMPUTED CONDITION: Bootstrapped AM: compute(curTrans.condition) = FALSE
UNMOTIVATED TRANSITION => TRUE
ON TRANSITION

EFFECT
   ACTION Bootstrapped AM: curTrans := out.next
STATE: Bootstrapped AM: Choose transition

LEAVING STATE: Bootstrapped AM: Choose transition
CHOSE TRANSITION FROM STATE: Bootstrapped AM: Choose transition
COMPUTED CONDITION: Bootstrapped AM: (curTrans is null) AND (start state is
history) = FALSE
COMPUTED CONDITION: Bootstrapped AM: curTrans is null = FALSE
UNMOTIVATED TRANSITION => TRUE
COMPUTED CONDITION: Bootstrapped AM: compute(curTrans.condition) = TRUE
ON TRANSITION

STATE: Bootstrapped AM: On transition

EFFECT
   ACTION Bootstrapped AM: DoEffect(curTrans.effect)
```

```
EFFECT
   ACTION decrease
LEAVING STATE: Bootstrapped AM: On transition
CHOSE TRANSITION FROM STATE: Bootstrapped AM: On transition
COMPUTED CONDITION: Bootstrapped AM: (curTrans.dest is final) AND
(curTrans.dest is exitPoint) AND (start state is history) = TRUE
ON TRANSITION

EFFECT
   ACTION Bootstrapped AM: exitPoint := curTrans.dest.name
   ACTION Bootstrapped AM: history := curState
STATE: Bootstrapped AM: The End

EFFECT
   ACTION Bootstrapped AM: currAuto := caller
LEAVING STATE: Bootstrapped AM: The End
CHOSE TRANSITION FROM STATE: Bootstrapped AM: The End
COMPUTED CONDITION: Bootstrapped AM: currAuto is null = FALSE
UNMOTIVATED TRANSITION => TRUE
ON TRANSITION

STATE: Bootstrapped AM: Leaving current state

EFFECT
   ACTION Bootstrapped AM: DoEffect(curState.exit)
EFFECT
   ACTION printLeaving
LEAVING STATE: Bootstrapped AM: Leaving current state
CHOSE TRANSITION FROM STATE: Bootstrapped AM: Leaving current state
UNMOTIVATED TRANSITION => TRUE
ON TRANSITION

EFFECT
   ACTION Bootstrapped AM: curTrans := out.first
STATE: Bootstrapped AM: Choose transition

LEAVING STATE: Bootstrapped AM: Choose transition
CHOSE TRANSITION FROM STATE: Bootstrapped AM: Choose transition
COMPUTED CONDITION: Bootstrapped AM: (curTrans is null) AND (start state is
history) = FALSE
COMPUTED CONDITION: Bootstrapped AM: curTrans is null = FALSE
COMPUTED CONDITION: exit point = next = TRUE
COMPUTED CONDITION: Bootstrapped AM: compute(curTrans.condition) = TRUE
ON TRANSITION

STATE: Bootstrapped AM: On transition

EFFECT
   ACTION Bootstrapped AM: DoEffect(curTrans.effect)
EFFECT
   ACTION increase
LEAVING STATE: Bootstrapped AM: On transition
CHOSE TRANSITION FROM STATE: Bootstrapped AM: On transition
COMPUTED CONDITION: Bootstrapped AM: (curTrans.dest is final) AND
(curTrans.dest is exitPoint) AND (start state is history) = FALSE
COMPUTED CONDITION: Bootstrapped AM: (curTrans.dest is final) AND
(curTrans.dest is exitPoint) = FALSE
COMPUTED CONDITION: Bootstrapped AM: (curTrans.dest is final) AND (start
state is history) = FALSE
COMPUTED CONDITION: Bootstrapped AM: curTrans.dest is final = FALSE
UNMOTIVATED TRANSITION => TRUE
ON TRANSITION

EFFECT
```

```
   ACTION Bootstrapped AM: curState := curTrans.dest
STATE: Bootstrapped AM: Current state

LEAVING STATE: Bootstrapped AM: Current state
CHOSE TRANSITION FROM STATE: Bootstrapped AM: Current state
COMPUTED CONDITION: Bootstrapped AM: curState is composite = TRUE
ON TRANSITION

STATE: Bootstrapped AM: GoTo

EFFECT
   ACTION Bootstrapped AM: DoEffect(curState.entry)
EFFECT
   ACTION printComeIn
   ACTION Bootstrapped AM: currAuto := curState.insAuto
   ACTION Bootstrapped AM: curState := find(owner.entryPoint)
LEAVING STATE: Bootstrapped AM: GoTo
CHOSE TRANSITION FROM STATE: Bootstrapped AM: GoTo
COMPUTED CONDITION: Bootstrapped AM: curState is null = TRUE
ON TRANSITION

EFFECT
   ACTION Bootstrapped AM: curState := start
STATE: Bootstrapped AM: Current state

LEAVING STATE: Bootstrapped AM: Current state
CHOSE TRANSITION FROM STATE: Bootstrapped AM: Current state
COMPUTED CONDITION: Bootstrapped AM: curState is composite = FALSE
COMPUTED CONDITION: Bootstrapped AM: curState is simple = FALSE
COMPUTED CONDITION: Bootstrapped AM: (curState is pseudo) AND (curState is
history) AND NOT (history is null) = TRUE
ON TRANSITION

EFFECT
   ACTION Bootstrapped AM: curState := history
STATE: Bootstrapped AM: Current state

LEAVING STATE: Bootstrapped AM: Current state
CHOSE TRANSITION FROM STATE: Bootstrapped AM: Current state
COMPUTED CONDITION: Bootstrapped AM: curState is composite = FALSE
COMPUTED CONDITION: Bootstrapped AM: curState is simple = TRUE
ON TRANSITION

EFFECT
   ACTION Bootstrapped AM: DoEffect(curState.entry)
EFFECT
   ACTION printRight
STATE: Bootstrapped AM: Leaving current state

EFFECT
   ACTION Bootstrapped AM: DoEffect(curState.exit)
LEAVING STATE: Bootstrapped AM: Leaving current state
CHOSE TRANSITION FROM STATE: Bootstrapped AM: Leaving current state
UNMOTIVATED TRANSITION => TRUE
ON TRANSITION

EFFECT
   ACTION Bootstrapped AM: curTrans := out.first
STATE: Bootstrapped AM: Choose transition

LEAVING STATE: Bootstrapped AM: Choose transition
CHOSE TRANSITION FROM STATE: Bootstrapped AM: Choose transition
COMPUTED CONDITION: Bootstrapped AM: (curTrans is null) AND (start state is
history) = FALSE
```

COMPUTED CONDITION: Bootstrapped AM: curTrans is null = FALSE
COMPUTED CONDITION: **IsLess** = TRUE
COMPUTED CONDITION: Bootstrapped AM: compute(curTrans.condition) = TRUE
ON TRANSITION

STATE: Bootstrapped AM: On transition

EFFECT
   ACTION Bootstrapped AM: DoEffect(curTrans.effect)
LEAVING STATE: Bootstrapped AM: On transition
CHOSE TRANSITION FROM STATE: Bootstrapped AM: On transition
COMPUTED CONDITION: Bootstrapped AM: (curTrans.dest is final) AND (curTrans.dest is exitPoint) AND (start state is history) = FALSE
COMPUTED CONDITION: Bootstrapped AM: (curTrans.dest is final) AND (curTrans.dest is exitPoint) = FALSE
COMPUTED CONDITION: Bootstrapped AM: (curTrans.dest is final) AND (start state is history) = FALSE
COMPUTED CONDITION: Bootstrapped AM: curTrans.dest is final = FALSE
UNMOTIVATED TRANSITION => TRUE
ON TRANSITION

EFFECT
   ACTION Bootstrapped AM: curState := curTrans.dest
STATE: Bootstrapped AM: Current state

LEAVING STATE: Bootstrapped AM: Current state
CHOSE TRANSITION FROM STATE: Bootstrapped AM: Current state
COMPUTED CONDITION: Bootstrapped AM: curState is composite = FALSE
COMPUTED CONDITION: Bootstrapped AM: curState is simple = TRUE
ON TRANSITION

EFFECT
   ACTION Bootstrapped AM: DoEffect(curState.entry)
EFFECT
   ACTION **decrease**
STATE: Bootstrapped AM: Leaving current state

EFFECT
   ACTION Bootstrapped AM: DoEffect(curState.exit)
LEAVING STATE: Bootstrapped AM: Leaving current state
CHOSE TRANSITION FROM STATE: Bootstrapped AM: Leaving current state
UNMOTIVATED TRANSITION => TRUE
ON TRANSITION

EFFECT
   ACTION Bootstrapped AM: curTrans := out.first
STATE: Bootstrapped AM: Choose transition

LEAVING STATE: Bootstrapped AM: Choose transition
CHOSE TRANSITION FROM STATE: Bootstrapped AM: Choose transition
COMPUTED CONDITION: Bootstrapped AM: (curTrans is null) AND (start state is history) = FALSE
COMPUTED CONDITION: Bootstrapped AM: curTrans is null = FALSE
COMPUTED CONDITION: **IsLess** = TRUE
COMPUTED CONDITION: Bootstrapped AM: compute(curTrans.condition) = TRUE
ON TRANSITION

STATE: Bootstrapped AM: On transition

EFFECT
   ACTION Bootstrapped AM: DoEffect(curTrans.effect)
LEAVING STATE: Bootstrapped AM: On transition
CHOSE TRANSITION FROM STATE: Bootstrapped AM: On transition

```
COMPUTED CONDITION: Bootstrapped AM: (curTrans.dest is final) AND
(curTrans.dest is exitPoint) AND (start state is history) = TRUE
ON TRANSITION

EFFECT
   ACTION Bootstrapped AM: exitPoint := curTrans.dest.name
   ACTION Bootstrapped AM: history := curState
STATE: Bootstrapped AM: The End

EFFECT
   ACTION Bootstrapped AM: currAuto := caller
LEAVING STATE: Bootstrapped AM: The End
CHOSE TRANSITION FROM STATE: Bootstrapped AM: The End
COMPUTED CONDITION: Bootstrapped AM: currAuto is null = FALSE
UNMOTIVATED TRANSITION => TRUE
ON TRANSITION

STATE: Bootstrapped AM: Leaving current state

EFFECT
   ACTION Bootstrapped AM: DoEffect(curState.exit)
EFFECT
   ACTION printLeaving
LEAVING STATE: Bootstrapped AM: Leaving current state
CHOSE TRANSITION FROM STATE: Bootstrapped AM: Leaving current state
UNMOTIVATED TRANSITION => TRUE
ON TRANSITION

EFFECT
   ACTION Bootstrapped AM: curTrans := out.first
STATE: Bootstrapped AM: Choose transition

LEAVING STATE: Bootstrapped AM: Choose transition
CHOSE TRANSITION FROM STATE: Bootstrapped AM: Choose transition
COMPUTED CONDITION: Bootstrapped AM: (curTrans is null) AND (start state is
history) = FALSE
COMPUTED CONDITION: Bootstrapped AM: curTrans is null = FALSE
COMPUTED CONDITION: exit point = next = FALSE
COMPUTED CONDITION: Bootstrapped AM: compute(curTrans.condition) = FALSE
UNMOTIVATED TRANSITION => TRUE
ON TRANSITION

EFFECT
   ACTION Bootstrapped AM: curTrans := out.next
STATE: Bootstrapped AM: Choose transition

LEAVING STATE: Bootstrapped AM: Choose transition
CHOSE TRANSITION FROM STATE: Bootstrapped AM: Choose transition
COMPUTED CONDITION: Bootstrapped AM: (curTrans is null) AND (start state is
history) = FALSE
COMPUTED CONDITION: Bootstrapped AM: curTrans is null = FALSE
COMPUTED CONDITION: exit point = end = TRUE
COMPUTED CONDITION: Bootstrapped AM: compute(curTrans.condition) = TRUE
ON TRANSITION

STATE: Bootstrapped AM: On transition

EFFECT
   ACTION Bootstrapped AM: DoEffect(curTrans.effect)
EFFECT
   ACTION printBye
LEAVING STATE: Bootstrapped AM: On transition
CHOSE TRANSITION FROM STATE: Bootstrapped AM: On transition
```

COMPUTED CONDITION: Bootstrapped AM: (curTrans.dest is final) AND (curTrans.dest is exitPoint) AND (start state is history) = FALSE
COMPUTED CONDITION: Bootstrapped AM: (curTrans.dest is final) AND (curTrans.dest is exitPoint) = FALSE
COMPUTED CONDITION: Bootstrapped AM: (curTrans.dest is final) AND (start state is history) = FALSE
COMPUTED CONDITION: Bootstrapped AM: curTrans.dest is final = TRUE
ON TRANSITION

EFFECT
   ACTION Bootstrapped AM: exitPoint := null
STATE: Bootstrapped AM: The End

EFFECT
   ACTION Bootstrapped AM: currAuto := caller
LEAVING STATE: Bootstrapped AM: The End
CHOSE TRANSITION FROM STATE: Bootstrapped AM: The End
COMPUTED CONDITION: Bootstrapped AM: currAuto is null = TRUE
ON TRANSITION

EFFECT
   ACTION Bootstrapped AM: terminate
GO FROM AUTO (id = 0) TO NOWHERE
TERMINATE!