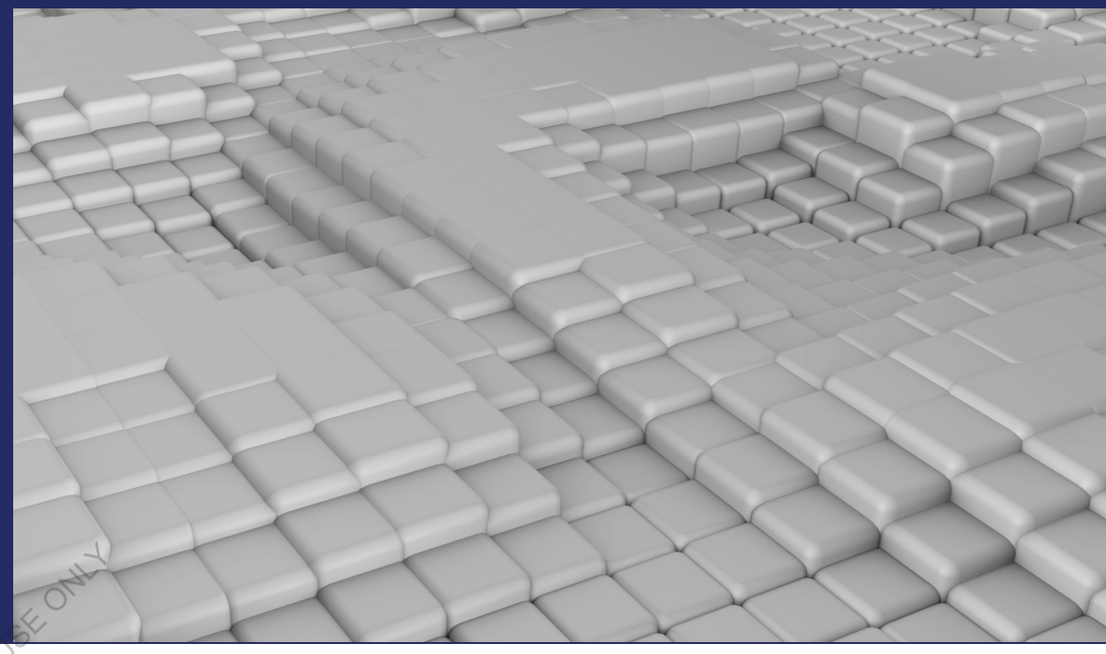
Приведены основные теоретические и практические сведения о системе PGEN++. Система позволяет: а) порождать решающие программы как по естественно-языковой так и по визуальной постановке задачи, б) реконструировать порождающие и/или иные смысловые модели программ/текстов, в) выполнять интеллектуальную трансформацию программ. Описаны порождающие объектно-событийные модели (ОСМ). Предложены распознающие ОСМ, позволяющие извлекать из текстовой постановки знания, и трансформировать их, порождая выходные модели, по которым генерируется программа. Разобрана проблема применения слоя грамматического разбора входных текстов, построен (на базе обычного/конструирующего XPath) Тьюринг-полный алгоритмический микроязык для взаимодействия с таким слоем. Для обработки извлеченных знаний используются скрипты обратного/прямого логического вывода. Предложена новая параллельная машина прямого смыслового вывода на базе системы слабых XPath-подобных ограничений, управляемая марковской моделью, матрица переходов которой синтезируется нейронной сетью на основе анализа прецедентов. Подходы апробированы в задачах порождения программ, верификации порождающих скриптов, распараллеливающей трансформации программ.

Пекунов Владимир Викторович родился 21 января 1977 года в г. Заполярный, Мурманской обл. Закончил с отличием ИГЭУ им В.И.Ленина. Доктор технических наук (2010). Член-корреспондент РАЕ (2019). Научные интересы: интеллектуальная обработка текстов, распараллеливающие трансформации программ, теоретические аспекты процессов обработки информации.

9 786203 029871

Система порождения программ PGEN++

Пекунов



Владимир Пекунов

Система порождения, реконструкции и преобразования программ PGEN++

LAP LAMBERT Academic Publishing

Владимир Пекунов

Система порождения, реконструкции и преобразования программ PGEN++

Владимир Пекунов

Система порождения, реконструкции и преобразования программ PGEN++

LAP LAMBERT Academic Publishing RU

Imprint

Cover image: www.ingimage.com

Publisher:
LAP LAMBERT Academic Publishing
is a trademark of
International Book Market Service Ltd., member of OmniScriptum Publishing Group
17 Meldrum Street, Beau Bassin 71504, Mauritius
Printed at: see last page
ISBN: 978-620-3-02987-1

Оглавление

Введение

Одной из важных задач в области искусственного интеллекта является, в частности, задача генерации решающей некоторую проблему программы по естественно-языковому описанию этой проблемы. Данная задача, которую при современном уровне развития технологий искусственного интеллекта вряд ли возможно решить универсально и в полном объеме, тем не менее может эффективно решаться системами порождения программ в целом ряде хорошо проработанных и формализованных областей (математические задачи, задачи проектирования стандартизованных технических объектов, различные типовые задачи программирования [в частности, генерация интерфейсов, генерация Web-контента, генерация подсистем для работы с базами данных и многие другие шаблонно решаемые задачи]). Поэтому, данная задача интересна не только теоретически, но и практически, и, несомненно, актуальна.

Существует два основных направления решения поставленной задачи: а) прямая трансформация постановочного текста на естественном языке в текст программы [25, 31] и б) многостадийная трансформация естественно-языковой постановки в программу (обязательным компонентом такой трансформации является выработка некоторой смысловой модели (например, плана решения задачи) [3, 22, 24]).

Первое направление может использовать продукционные правила в сочетании с нейросетями, определяющими сравнительные вероятности применения того или иного правила (как это сделано в работе [31]), или прибегать к помощи технологии Statistical Machine Translation (SMT), которая определяет статистически лучшие варианты прямой трансляции отдельных фрагментов входной постановки в фрагменты выходной постановки, объединяет их и проверяет результат по некоторой языковой модели [25]. При всей интересности данного направления нельзя не признать, что оно более перспективно для решения достаточно локальных задач, таких как автокомментирова-

ние программного кода или машинный перевод, выполняемый по отдельным предложениям. Задача же генерации программы может требовать комплексного анализа смысла исходного текста в целом.

Поэтому, более продуктивным представляется второе направление. В его рамках возможны различные подходы к построению смысловой модели. Например, в системе IPGS [3] используются многоленточные машины Тьюринга в сочетании с элементами теории формальных грамматик. Заметим, что данный подход, скорее, более применим для решения несколько иной задачи (см., например, [29]), а именно – для извлечения смысловой модели из такого четко формализованного текста, как текст программы, то есть, например, для реконструкции порождающей модели программы из результата порождения (например, в целях верификации порождающих скриптов). Для текстов же на естественном языке подходы такого рода не очень просты в реализации и могут иметь достаточно ограниченное применение.

Более перспективным представляется подход, при котором используются (в той или иной форме) хорошо известные схемы определения грамматических связей между отдельными словами [21], даже на основании первичного анализа которых уже можно делать некоторые выводы о содержании поставленной задачи. Такой подход успешно представлен, например, в работах [22, 24]. Однако нельзя не отметить общий недостаток этих работ – крайнюю простоту получаемой смысловой модели и механизма, реализующего генерацию кода (используются специальные таблицы трансляции). Вызывает определенное сомнение пригодность подобной схемы для существенно более сложных случаев (нежели разобранные в упомянутых работах), особенно требующих дополнения выделенной смысловой модели (которая может быть представлена в крайне неполном, возможно, чисто декларативном виде) до полного плана решения задачи.

Представляется целесообразным построение смысловой модели за два этапа. На первом этапе необходимо провести лексико-синтаксический разбор исходного текста на естественном языке с разбивкой его на предложения, с выявлением отдельных элементов пред-

ложений и, возможно, с анализом грамматических связей между этими элементами. В результате получаем первичную фактологическую модель текста, включающую набор высказываний, каждое из которых может быть описано объектом некоторого класса заданной предметной области (схожий подход применен в [24]). Каждый класс представляет либо некоторую сущность, либо действие, либо отношение, а поля соответствующего объекта такого класса содержит, соответственно, атрибуты сущности, объект/субъект действия, параметры отношения.

На втором этапе построения смысловой модели выявленный набор первичных объектов анализируется (уже на уровне отношений между различными объектами) и достраивается до полной модели, содержащей уже все необходимую для решения поставленной задачи информацию. Данный процесс достаточно нетривиален алгоритмически, поэтому его целесообразно решать в некоторой формально-логической постановке.

Здесь возможны два основных подхода – *прямой и обратный логический вывод*. Обратный вывод дает требуемый результат за сравнительно небольшое время, его целесообразно использовать в случаях, когда четко ясна структура целевого утверждения. В нашем случае это, фактически, означает, что должны быть заранее ясны структура и состав конечной смысловой модели. Такая ясность более характерна для решения вышеупомянутых задач реконструкции моделей программ (текст работающей программы является абсолютно достаточным для решения задачи восстановления смысла, не так сложно записать цель и четкие схемы обобщения исходной фактологической модели, поскольку, обычно, отсутствуют «пропущенные» элементы модели, которые требовалось бы восстановить). Тем не менее, в не очень сложных случаях возможно применение обратного логического вывода и для восполнения первичных моделей, полученных из текстов на естественном языке, это также будет продемонстрировано в данной работе.

На практике все же более целесообразен прямой логический вывод, для которого задается не целевое утверждение, а некоторые наборы ограничений процесса построения и наборы тестов, через которые должна успешно проходить программа, сгенерированная по очередному (формально корректному) варианту плана решения задачи. Такой вывод обычно более трудоемок, чем обратный, но допускает чистую декларативность и «отрывочность» логических правил, а его трудоемкость может быть скомпенсирована разработкой специальных распараллеленных машин прямого логического вывода.

Заметив, что эффективное программирование логического вывода (особенно прямого) является достаточно нетривиальной задачей даже для программиста, а пользователями системы порождения программ преимущественно являются непрограммирующие специалисты по знаниям в определенных предметных областях, целесообразно максимально упростить форму представления смысловой модели (как для прямого, так и для обратного вывода), которая определила бы и средства ее элементарного анализа. Что же касается логического достраивания модели, то в случае обратного логического вывода достаточно адекватен классический предикативный синтаксис. Однако предикативная запись не так проста в случае прямого логического вывода – здесь, видимо, целесообразно применение какого-то более простого и близкого к выбранной форме представления смысловой модели синтаксиса.

Адекватной формой представления смысловой модели может быть XML-документ, набор тэгов которого способен содержать описания не только объектов, входящих в план решения задачи, но и отношений между ними. В связи с этим логично выбрать в качестве базового средства анализа документа какой-либо из стандартных формальных языков обработки XML-документов, который в случае прямого логического вывода мог бы стать и адекватным средством достраивания модели. В частности, достаточно интересным представляется выбор языка запросов XPath, причем существенный (не только теоретический, но и практический) интерес представляют его модифика-

ции, позволяющие применить основанный на XPath синтаксис для конструирования и описания небольших простых алгоритмов, носящих вспомогательный характер.

В данной работе задача порождения программ по исходным естественно-текстовым постановкам рассматривается на примере *системы PGEN++* [6, 11], которая изначально поддерживала порождение произвольных программ по визуальной модели с развитыми элементами логического программирования, выгодно отличающими данную систему от многих иных систем[1] полной автоматизации программирования (ДРАКОН [5], Draco [17], TAMPR [20], TXL [17], системы В.Д.Ильина [4], различных реализаций DSL-подходов [19]). Как будет показано далее, система PGEN++ была модифицирована, в частности была введена возможность последовательного или сканирующего распознавания (индукции) смысла простых текстов на естественном языке (с построением смысловой объектно-событийной модели (ОСМ), ее возможной последующей реструктуризацией [на базе прямого или обратного логического вывода] и генерацией выходной программы) или алгоритма программы. Поскольку данные возможности реализуются с помощью целого набора микроязыков, то, несмотря на их простоту, написание соответствующих скриптов и шаблонов требует существенных временных затрат. Поэтому возникает задача частичной автоматизации их разработки, по крайней мере в части, относящейся к группам регулярно-логических выражений и скриптам прямого логического вывода. Иными словами, возникает задача приобретения знаний об индукции и трансформации ОСМ.

Относительно несложно построить систему, решающую такую задачу для какой-либо определенной предметной области [11], значительно сложнее построить универсальную систему, способную приобретать знания о решении конкретной проблемы в произвольной предметной области [28]. Можно выделить четыре основных подхода к

[1] Единственными иными, известными автору системами, в той или иной степени поддерживающими логическое программирование, являются ПРИЗ и IPGS [3].

приобретению знаний в системах автоматической генерации программного кода: а) ручное составление правил вывода/восполнения модели задачи, б) автоматическое составление правил на основе некоторого набора обучающих пар «постановка задачи – программа» (может быть определен вручную [31] или собран на каких-либо интернет-ресурсах, подобных GitHub или StackOverflow [22, 30]), в) прямое применение обучающего набора (без вывода правил) и г) комбинированный. Первый подход [24] достаточно надежен, но требует участия человека и весьма затратен по времени. Второй подход [22] участия человека не требует и показывает достаточно хорошие (однако более слабые в сравнении с первым подходом) по качеству результаты. Третий подход [25, 31] также не требует участия человека, однако не вполне детерминирован и, как следствие, требует достаточно совершенных схем контроля результата (см., например, [25]), сложных как с синтаксической, так и с семантической точки зрения. Наиболее перспективным, вероятно, является четвертый подход (см., например, [28]), позволяющий сочетать достоинства различных вариантов и, вероятно, являющийся наиболее уместным для систем класса DSL (Domain Specific Language), к которым относится и PGEN++. В его рамках изначально присутствуют четкие формальные правила (А) вывода конечного кода (созданные вручную) и требуется лишь определить механизмы генерации правил (Б) трансформации естественно-языковых описаний задачи во входные DSL-описания, что является существенно более простой задачей, чем построение механизмов генерации правил трансляции естественно-языковых описаний непосредственно в программный код [24, 25, 31]. Такой подход потенциально способен выдать достаточно качественные результаты, нередко полностью лишенные ошибок.

На основании данного краткого анализа было принято решение строить механизм приобретения знаний в системе порождения программ PGEN++ (фактически являющейся DSL-системой) на базе четвертого подхода. При этом правила вывода кода из смысловой модели программы (ОСМ) составляются вручную, а правила преобразования

естественно-языковой постановки в смысловую модель, основанные на прямом логическом выводе, генерируются автоматически, в процессе приобретения знаний.

В связи с вышеуказанными модификациями системы PGEN++, автором было принято решение полностью опубликовать в настоящей работе основные известные ранее и новые теоретические формализмы, принятые в данной системе, а также основные практические сведения, необходимые для работы с ней, включая краткие руководства пользователя.

Целью данной работы является повышение функциональности схем порождения программ по естественно-языковой постановке задачи, а также схем трансформации (прямой или опосредованной через распознавание/порождение) программ/текстов.

Для достижения данной цели поставим следующие *задачи*:

а) предложить теоретические основы распознавания, первичной обработки и порождения текстов на естественных и искусственных языках с получением первичных смысловых XML-моделей в системе порождения программ PGEN++ [7, 12], которая работает с четко формализованными предметными областями;

б) предложить подход к достраиванию полученных первичных смысловых моделей на базе обратного логического вывода;

в) предложить подход к достраиванию полученных первичных смысловых моделей на базе прямого логического вывода с привлечением XPath-технологий;

г) разработать адекватную, автоматически настраивающуюся на задачу машину прямого логического вывода;

д) разработать адекватные языковые средства (языки описания, обобщенные языки, а также множество XPath-подобных языков – конструирующих, алгоритмических, семантических на базе слабых ограничений);

е) предложить подходы к частичной автоматизации процессов разработки скриптов и шаблонов, используемых для решения задач распознавания и трансформации/достраивания ОСМ;

ж) провести испытания разработанных программных средств на типовых задачах: порождения программ по исходным текстам на естественном языке с применением механизмов как прямого, так и обратного логического вывода; реконструкции исходной порождающей XML-модели по порожденной программе в целях верификации порождающих скриптов; трансформации программ, в частности автоматического распараллеливания.

Краткое руководство пользователя по работе с системой PGEN++ дано в приложениях (Приложение П1, Приложение П2, Приложение П3, Приложение П4).

Глава 1. Система PGEN++. Общие сведения. Порождение программ

Целью данной главы является рассмотрение теоретических и практических основ представления данных и порождения программ в системе PGEN++, основанной на объектно-событийных моделях (ОСМ). Для реализации данной цели поставим следующие задачи:

а) дать формальное описание ОСМ и процесса генерации программ/текстов на их основе, включая практические сведения о программировании решающих и порождающих скриптов;

б) рассмотреть проблему представления алгоритмов и планов решения задачи в виде ОСМ.

1.1. Общее понятие об объектно-событийных моделях (ОСМ) порождения программ

В базовом варианте ОСМ представляет собой сеть связанных объектов, относящихся к некоторой иерархии классов рассматриваемой предметной области, описывающих поставленную задачу. Элементы задачи (объекты) и связи между ними обычно отображаются в визуальную блочную диаграмму. В процессе интерпретации модели объекты срабатывают покаскадно, при наступлении каждого очередного события (ведется календарь событий, который может быть пополнен в любой момент любым из объектов модели). Начальный календарь событий содержит некоторое заранее определенное множество событий. В ходе реакции на события объекты обмениваются данными по связям, а также через единую базу данных «почтовый ящик», представляющую собой ассоциативный контейнер, элементами которого могут быть структуры произвольной сложности. При реакции на событие каждый объект выполняет некоторый алгоритм, представляющий собой метод обработки данного события (в частности, дан-

ный метод может представлять собой алгоритм, генерирующий некий код, или быть любым иным специализированным методом.

При всей простоте такой концепции, в работах автора показано [14], что ОСМ равномощна машине Тьюринга, следовательно, может реализовать любой вычислимый по Тьюрингу алгоритм.

В исходном варианте системы PGEN++ объектно-событийная модель может использоваться как для описания задачи, так и для порождения программы. Дадим ее более формальную постановку.

1.1.1. Формальное описание ОСМ

Модель представлена графом (P, E), где P — множество узлов (объектов различных классов), E — множество дуг (связей между объектами). На множестве P определим функцию принадлежности к классу: class: $P \to C$, где C — множество классов. Классы C делятся на *две базовые категории*: решающие C_L и порождающие C_G, причем $C = C_L \cup C_G$.

Решающие классы являются, по сути, генерирующими по отношению к модели: они обладают двумя логическими методами-скриптами (на языке GNU Prolog[1]), содержащими, соответственно, инициализирующий и решающий предикаты. Каждый предикат имеет входные (уже связанные) и выходные (свободные) переменные, состав которых определяется наборами входов и выходов, указанных в декларации соответствующего класса.

Порождающие классы являются генерирующими по отношению к результирующему тексту программы: они содержат порождающие методы на языке PHP, создающие выходной текст в зависимости от текущего состояния модели, полученного в ходе ее интерпретации.

[1] Подробная информация о возможностях языка содержится на сайте: http://www.gprolog.org

Существует два основных типа связей между объектами: основные и вспомогательные. Основные связи определяют порядок срабатывания объектов и являются «каналами» передачи данных. Вспомогательные связи — лишь констатация факта наличия связи, отражающей структурные аспекты модели. Связи имеют различную семантику: «используется для», «входит в», «формирует», «имеет отображение» и другие. Любая связь исходит из выходного контакта объекта и входит во входной контакт объекта.

Граф модели может содержать циклы, но каждый цикл должен содержать хотя бы одну вспомогательную связь. Подграф, построенный лишь на основных связях, представляет собой сеть — циклы по основным связям не допускаются.

Класс S является (см. рис. 1.1) пятеркой (N_c, I, O, F, M), где N_c — идентификатор (имя) класса, I и O — наборы входных и выходных контактов (определим функциями I = in(S), O = out(S)), F — поля, M — порождающие методы. Отношение иерархии parent: $C \to C$, задано функцией

$$\forall a \in C \;\; \exists \text{parent}(a) = \begin{cases} (\varepsilon, \varnothing, \varnothing, \varnothing, \varnothing), & \text{если a есть корень иерархии;} \\ c \in C, & \text{если c есть предок a,} \end{cases}$$

где ε — пустая цепочка.

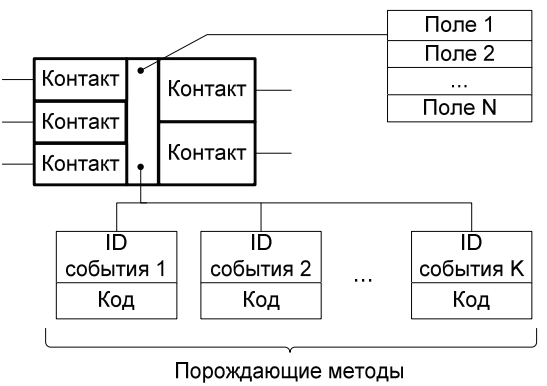


Рис. 1.1. Схематическое изображение класса

Контакты, поля и методы любого класса могут быть собственными, унаследованными или переопределенными. Любой контакт $Z \in \text{in}(S) \cup \text{out}(S)$ класса S является шестеркой (N_t, T, L, Min, Max, D), где N_t — идентификатор контакта, T ∈ {входной, выходной} — тип контакта, L — множество допустимых выходных связей:

$$L = \{(s,z) | Z \in \text{out}(S) \,\&\, s \in C \,\&\, z = \text{in}(s) \,\&\, [\text{допустима связь}(S,Z) \rightarrow (s,z)]\},$$

Min ∈ {0, 1} — минимальная степень контакта, Max ∈ {1, ∞} — максимальная степень, D = D[N_k] — ассоциативный кортеж (буфер передачи данных), отражающий состояние контакта объекта при порождении программы, причем N_k — имя k-й ячейки кортежа.

Множество полей F класса S можно также представить в виде ассоциативного кортежа F = F[N_j], хранящего текущее состояние объекта, принадлежащего классу S, причем N_j — имя j-й ячейки кортежа.

В множестве M класса S каждый порождающий метод является программным скриптом, реализующим реакцию на некоторое событие, произошедшее в процессе интерпретации модели. Удобно представить данное множество в виде вектора M = (M_{S1}, M_{S2}, … M_{SN}), где N — размер календаря событий. Реакция на событие может включать действия четырех типов: а) контекстную генерацию фрагмента кода на основании значений полей F и информации, поступившей на входные контакты (хранящейся в соответствующих кортежах D), б) планирование нового события (включение его в календарь событий), в) изменение значений в ассоциативных кортежах D выходных контактов, г) изменение значений в *глобальном ассоциативном кортеже* B — «почтовом ящике», предоставляющем дополнительные *прямые средства обмена данными* между объектами.

Отметим, что для удобства представления модели в виде блочной схемы существуют специальные *классы-контейнеры*, представляющие «обертку» для произвольного разработанного ранее фрагмента модели (P^*, E^*). При интерпретации модели контейнеры непосредственно заменяются фрагментами (P^*, E^*), включаемыми в общую модель (P, E).

1.1.2. Формальное описание интерпретации ОСМ

Интерпретация модели, в общем случае, проводится в два этапа:

1. *Трансформирующая интерпретация*. Если модель содержит объекты решающих классов, то к ней применяется двухстадийная модифицирующая процедура логического вывода. Формируются два целевых предиката, требующих, соответственно, последовательного доказательства инициализирующих и решающих предикатов, входящих в состав методов объектов решающих классов.

Инициализирующие предикаты анализируют постановку задачи и подготавливают данные для решения, заполняя базу фактов, а *решающие предикаты* дедуцируют план решения или фрагмент постановки задачи и *вводят его в модель* в виде соответствующей подсхемы, представляющей собой совокупность взаимосвязанных объектов. Для предикатов структура и параметры задачи описаны в виде динамического набора фактов, который имеет однозначное XML-представление.

Последовательность доказательства определяется структурой связей модели (по методике, аналогичной сетевому графику работ). Если результирующее описание задачи (модель) вновь содержит объекты решающих классов, то оно повторно рассматривается по вышеуказанной схеме и так далее.

Таким образом, здесь утверждения, вытекающие из структуры ОСМ, согласуются с базой фактов, в число которых входят непосредственно сведения о структуре ОСМ, а также часть результатов исполнения ОСМ, представляющих собой промежуточные или конечные результаты описанной унификации. Данный процесс осуществляется специальной машиной вывода ОСМ согласно алгоритму, описанному в работе [14].

2. *Содержательная интерпретация*. Осуществляется для модели-схемы, состоящей исключительно из объектов порождающих классов, на двух взаимосвязанных уровнях, которые условно назовем событийным и объектно-сетевым. Схематически процесс интерпретации

показан на рис. 1.2, где окружностями с цифрами обозначены объекты модели, пустыми квадратами — методы, штрихованными прямоугольниками — фрагменты программы, порожденные соответствующими методами.

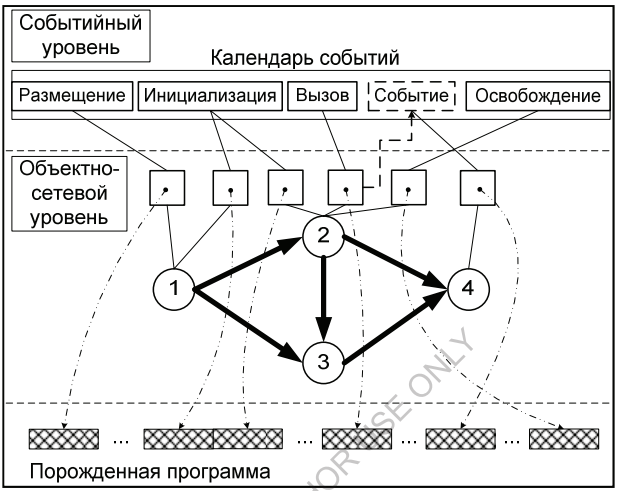


Рис. 1.2. Схема процесса интерпретации модели и порождения программы

На событийном уровне процесс интерпретации управляется календарем событий, представленным вектором $C = (C_1, C_2, \ldots C_N)$. В самом начале интерпретации календарь содержит четыре обязательных события, соответствующие основным этапам работы с любым ресурсом: размещение, инициализацию, вызов, деинициализацию. Произошедшее событие интерпретируется на объектно-сетевом уровне, после чего происходит следующее событие. Интерпретация модели заканчивается, если исчерпан календарь событий.

В процессе интерпретации i-го события (объектно-сетевой уровень) используется методика, подобная той, которая используется для сетевых графиков работ. Покаскадно срабатывают группы объектов модели. Сначала срабатывают объекты, не имеющие входных контактов ($I = \varnothing$), а также объекты, к входным контактам которых не под-

соединены основные связи. При срабатывании j-го объекта активизируется его метод $(M_j)_i$, ответственный за обработку i-го события, который генерирует фрагмент программы (символьную цепочку) $(Z_j)_i$, формирует данные на своих выходах и, возможно, планирует новые события. Все прочие объекты срабатывают тогда и только тогда, когда сработают все объекты, соединенные по основным связям с их входными контактами.

При выполнении порождающих методов объект может поместить произвольный массив значений в любую ячейку кортежей $D[N_k]$ своих выходных контактов и прочитать массив значений из любой ячейки $D[N_k]$ входных контактов. Предусмотрена возможность *исполнения запросов к структуре модели* (как это иногда осуществляется в различных технологиях класса DSL), что в ряде случаев может снижать трудоемкость ее анализа. При этом модель отображается в XML-документ, а запросы пишутся на языке XPath, что вполне отвечает исходным установкам настоящей работы, данным во введении.

Интерпретация события заканчивается, когда сработают все объекты модели. Последовательность срабатываний объектов может быть описана вектором Q, элемент q_k которого содержит номер объекта, сработавшего k-м по счету. Если несколько объектов сработали одновременно, то их номера располагаются в векторе Q подряд и их последовательность не имеет значения.

По окончании интерпретации i-го события формируется символьная цепочка

$$Y_i = \left(Z_{q_1}\right)_i + \left(Z_{q_2}\right)_i + \ldots + \left(Z_{q_n}\right)_i,$$

где n — число объектов модели, знаком «+» обозначено слияние цепочек.

Порожденная программа G определяется символьной цепочкой

$$G = Y_1 + Y_2 + \ldots + Y_N,$$

где N — общее количество произошедших событий.

Описанные модель и схема ее интерпретации позволяют избежать неоднозначностей порядка компоновки символьных цепочек, связанных с циклическим срабатыванием элементов модели. Поддер-

живается возможность неоднократного срабатывания объектов за счет ввода событийной составляющей, что позволяет *добиться порождения одним объектом фрагментов кода, чередуемых с фрагментами*[1], *порожденными другими объектами*.

1.1.3. Трансляция

Сначала всегда производится интерпретация модели, результатом которой является программа на конкретном (например, Pascal, C, язык M-скриптов системы MATLAB) или специальном обобщенном алгоритмическом языке. В первом случае получаем готовую к исполнению программу, во втором — необходима дополнительная трансляция обобщенной программы на конкретный язык программирования (схожий подход применен в работе [18]). При всей простоте первого подхода нельзя не отметить его серьезный недостаток: порождающие методы ориентируются на конкретный язык. Если возникает потребность в генерации программы на ином языке, то приходится вносить изменения в реализацию методов, что требует существенных временных затрат. Второй подход является *более гибким* (специфические конструкции языка генерируются транслятором), но более сложным, поскольку возможности обобщенного языка существенно ограничены именно в силу его общности. Однако при этом разработка порождающих методов производится однократно, что уменьшает временные затраты.

Транслятор с обобщенного алгоритмического на конкретный язык программирования написан на языке SNOBOL4+, ориентированном на обработку строк, позволяющем с легкостью проводить лексико-синтаксический анализ и компиляцию.

[1] Чередуемость генерируемого кода – важное свойство порождающих ОСМ. Если применить инверсию, распознающие ОСМ, базирующиеся на тех же принципах, могут быть способны к распознаванию сложного входного текста, изначально включающего чередующиеся фрагменты, что и будет продемонстрировано нами далее.

Обобщенный алгоритмический язык достаточно прост и не требует детального описания синтаксиса, в следующем пункте он будет проиллюстрирован на примере простой программы, являющейся результатом трансляции модели, описывающей решение системы из двух дифференциальных уравнений методом Эйлера.

1.2. Пример программы на обобщенном алгоритмическом языке и результата трансляции

Программа на обобщенном алгоритмическом языке:

```
#DEFVAR(#CHAR,Map)
#DEFVAR(#FILE,Map_File)
#DEFARRAYVAR(#FLOAT,U,2)
#DEFVAR(#INT,cntU)
#PROCEDURE U_Init(#CHAR,Map;#FLOAT,#BYREF(U))
  #SELECT(Map)
    #SELECTOR('A')
      #SET(#DEREF(U),1.0)
    #SELECTOR(#ELSE)
      #SET(#DEREF(U),0)
  #ENDSELECT
#ENDPROCEDURE
#DEFVAR(#FLOAT,Func1_Result)
#PROCEDURE Func1(#FLOAT,U;#FLOAT,#BYREF(Func1_Result))
  #SET(#DEREF(Func1_Result),-U)
#ENDPROCEDURE
#DEFARRAYVAR(#FLOAT,U1,2)
#DEFVAR(#INT,cntU1)
#PROCEDURE U1_Init(#CHAR,Map;#FLOAT,#BYREF(U1))
  #SELECT(Map)
    #SELECTOR('A')
      #SET(#DEREF(U1),2.0)
    #SELECTOR(#ELSE)
      #SET(#DEREF(U1),0)
  #ENDSELECT
#ENDPROCEDURE
#DEFVAR(#FLOAT,Func_Result)
```

```
#PROCEDURE Func(#FLOAT,U;#FLOAT,U1;#FLOAT,#BYREF(Func_Result))
  #SET(#DEREF(Func_Result),U+U1)
#ENDPROCEDURE
#PROCEDURE Euler(#FLOAT,F; #FLOAT,#BYREF(Fn); #FLOAT,G; #DOUBLE,
TAU)
  #SET(#DEREF(Fn),F+TAU*G)
#ENDPROCEDURE
#PROGRAM
  #DEFVAR(#DOUBLE,EndTime)
  #DEFVAR(#DOUBLE,TAU)
  #DEFVAR(#DOUBLE,Time)
  #FILEOPEN(Map_File,'SMap.map')
  #FILEREAD(Map_File,#REF(Map),#SIZE(#CHAR))
  #CALL U_Init(Map,#REF(U[#LOWDIM]))
  #CALL U1_Init(Map,#REF(U1[#LOWDIM]))
  #SET(EndTime,0.01)
  #SET(TAU,0.000001)
  #SET(Time,0)
  #WHILE(Time #LT# EndTime)
    #SET(cntU,#LOWDIM)
    #CALL Func1(U[#LOWDIM],#REF(Func1_Result))
    #SET(cntU1,#LOWDIM)
    #CALL Func(U[#LOWDIM],U1[#LOWDIM],#REF(Func_Result))
    #CALL Euler(U[cntU],#REF(U[cntU+1]),Func_Result,Time)
    #INCREMENT(cntU,1)
    #CALL Euler(U1[cntU1],#REF(U1[cntU1+1]),Func1_Result,Time)
    #INCREMENT(cntU1,1)
    #SET(U[#LOWDIM],U[cntU])
    #SET(U1[#LOWDIM],U1[cntU1])
    #SET(Time,Time+TAU)
  #ENDWHILE
  #FILECLOSE(Map_File)
#ENDPROGRAM
```


Соответствующая программа на языке C имеет вид:

```
char Map;
FILE * Map_File;
float U[2];
int cntU;
void U_Init(char Map,float * U) {
  switch(Map) {
    case 'A' :
      *U = 1.0;
    break;
```

```
    default :
      *U = 0;
  break;
  }
}
float Func1_Result;
void Func1(float U,float * Func1_Result) {
  *Func1_Result = -U;
}
float U1[2];
int cntU1;
void U1_Init(char Map,float * U1) {
  switch(Map) {
    case 'A' :
      *U1 = 2.0;
    break;
    default :
      *U1 = 0;
  break;
  }
}
float Func_Result;
void Func(float U,float U1,float * Func_Result) {
  *Func_Result = U+U1;
}
void Euler(float F,float * Fn,float G,double TAU) {
  *Fn = F+TAU*G;
}
void main(int argc, char ** argv) {
  double EndTime;
  double TAU;
  double Time;
  Map_File = fopen("SMap.map","r");
  fread(&Map,sizeof(char),1,Map_File);
  U_Init(Map, &U[0]);
  U1_Init(Map, &U1[0]);
  EndTime = 0.01;
  TAU = 0.000001;
  Time = 0;
  while (Time < EndTime) {
    cntU = 0;
    Func1(U[0], &Func1_Result);
    cntU1 = 0;
```

```
    Func(U[0], U1[0], &Func_Result);
    Euler(U[cntU], &U[cntU+1], Func_Result, Time);
    cntU++;
    Euler(U1[cntU1], &U1[cntU1+1], Func1_Result, Time);
    cntU1++;
    U[0] = U[cntU];
    U1[0] = U1[cntU1];
    Time = Time+TAU;
  }
  fclose(Map_File);
}
```

1.3. Архитектура распределенной подсистемы порождения программ

Следует заметить, что машина вывода моделей в общем случае работает по достаточно сложной схеме [14], предусматривающей порождение новых машин вывода для решения подзадач в асинхронном режиме. Эти машины содержатся в очереди Q родительской машины и могут либо последовательно исполняться на одном компьютере, либо передаваться на параллельную обработку на иные компьютеры. Таким образом, архитектура системы является распределенной. Была выбрана схема, в которой любой из экземпляров системы порождения программ может выполнять как функции сервера, обеспечивающего взаимодействие с пользователем и распределение вычислительных ресурсов, так и функции клиента (исполнителя порожденных заданий).

Роль конкретного экземпляра системы определяется динамически, по факту очередности запуска: первый экземпляр берет на себя функции сервера, остальные — клиентов. Основная часть заданий обычно генерируется сервером, однако возможны и ситуации, когда в процессе исполнения заданий на клиенте генерируется новая их серия (имеется в виду случай, когда дочерняя машины вывода породила еще одну или несколько машин). В таком случае клиентские экземпляры системы порождения берут на себя часть серверных функций (засылка

задания со всеми необходимыми файлами и ожидание результата его обработки) по отношению к порожденным заданиям, за исключением распределения вычислительных ресурсов — им занимается лишь серверный экземпляр системы. Он и предоставляет (по поступающим запросам) клиентские экземпляры системы в качестве исполнителя порожденных машин вывода.

Данная идеология работы определила и механизмы реализации управляющей выводом части системы порождения программ. Каждый экземпляр системы является многопоточным приложением, в котором *постоянно работает не менее трех потоков*: основного, управляющего и очереди заданий.

Основной поток содержит машину вывода и генерирует для сервера сообщение об окончании работы машины (то есть об освобождении узла-экземпляра системы) и об окончании исполнения задания.

Управляющий поток реализует обработку запросов, поступающих от других экземпляров системы. В начале работы он посылает групповой запрос (IP Multicast) узлам локальной сети, чтобы определить, является ли текущий экземпляр первым из запущенных, то есть серверным, или же клиентским. После этого поток переходит в режим генерации и обработки запросов по локальной сети, преимущественно касающихся занятости узлов и их выделения. Он также принимает задания для обработки от серверного экземпляра системы.

Поток, реализующий очередь заданий, также запускается при старте системы. Он отслеживает занятость текущего экземпляра системы заданием, сообщая серверу об его освобождении, ведет список генерируемых в системе заданий, запрашивает свободные узлы (экземпляры системы) для отсылки задания и ведет их список (это относится в первую очередь к серверу), при появлении свободного узла отсылает ему задание со всеми необходимыми файлами.

При отсылке каждого задания для него также порождается *отдельный поток*, который устанавливает связь с удаленным компьюте-

ром, на котором обрабатывается задание, и переходит в режим ожидания результатов.

1.4. Практические аспекты реализации порождающих методов

Не ставя перед собой задачу исчерпывающего технического описания процессов разработки порождающих (а далее – и решающих) методов, автор, все же решил прояснить их отдельные практические аспекты.

Все классы системы описаны совокупностью различных файлов, находящихся в иерархической структуре подпапок папки **Classes**, находящейся в рабочем каталоге системы. Структура упомянутых подпапок и определяет структуру иерархии классов, в каждой папке находятся файлы, описывающие класс, идентификатор которого совпадает с именем папки.

Порождающие методы классов находятся в скриптовых файлах **script.php3**. Каждый такой файл должен содержать законченный скрипт на языке PHP.

На этапе содержательной интерпретации модели система формирует в каталоге системы общий порождающий скрипт **_.php3** для текущей модели и отправляет его на исполнение в интерпретатор PHP. Выходной поток, генерируемый таким скриптом, воспринимается как результат содержательной интерпретации (обычно это порожденная программа).

При автоматическом формировании общего скрипта широко используется классическое объектно-ориентированное программирование. В скрипт вписываются все классы, содержащиеся в модели, все их предки выше по иерархии, а также код, создающий экземпляры порождающих классов (объекты модели) и организующий цикл обработки событий модели. В цикле обработки событий на каждой итерации вызываются методы **Generate** объектов модели в порядке, соот-

ветствующем идее сетевого графика работ, узлами которого являются объекты модели, после чего вызывается стандартная функция **eval**, интерпретирующая ленту системы **$Tape** (если она не пуста).

1.4.1. Разработка классов

Новый класс вводится в систему путем создания некоторой подпапки (в папке **Classes**), имя которой совпадает с идентификатором класса (обычно должно использоваться умолчание, что все имена классов начинаются с префикса **cls** – это важно для успешной работы некоторых подсистем). При этом цепочка родительских папок, фактически, определяет цепочку наследования свойств, методов и контактов классов. Главным файлом, описывающим класс, является **Class.ini**. Обычно он сопровождается, как минимум, файлом **script.php3**, о котором шла речь выше, а также, если класс является решающим, файлами **init.pl** и **solve.pl**, о которых будет сказано далее. Также может присутствовать файл иконки класса **image.jpg**, а в случае, если для класса разработаны индуцирующие методы, то и файлы **induct.mac**, **induct.im**, **induct.ini**, **induct.log** (некоторые из этих файлов рассматриваются в приложениях настоящей работы).

Файл **Class.ini** может содержать следующие секции и переменные:

1. Обязательная секция **[Definition]**:

а) переменная **Name** должна содержать имя класса;

б) переменная **InheritScript** должна содержать логическое значение (0/1), определяющее, будет ли порождающий метод данного класса в своем начале вызывать унаследованный метод.

2. Обязательная секция **[Parameters]**, содержащая определения дополнительных параметров, вводимых данным классом. Каждый параметр определяется небольшой грамматикой:

параметр = спецификация идентификатор_параметра «**=**» значение_по_умолчанию

спецификация = «{» [флаги] «**;**» [тип] «**;**» название «}»

флаги = флаг {«**+**» флаг}

флаг = неизменяемый | обязательный | скрытый | уникальное_значение

неизменяемый = «**Lock**»

обязательный = «**Required**»

скрытый = «**Hide**»

уникальное_значение = «**Unique**»

тип = список_объектов | селектор | строка | многострочный_текст

многострочный_текст = «**Text**»

строка = «**Input**»

список_объектов = «**List**» «(» список_классов «**;**» разделитель «)»

список_классов = идентификатор_класса {«**,**» идентификатор_класса}

разделитель = «**'**» строка_текста «**'**»

селектор = «**Selector**» «(» вариант {«**,**» вариант} «)»

вариант = «**'**» строка_текста «**'**»

название = «**'**» строка_текста «**'**»

Если в спецификации параметра не указаны флаги, то он считается необязательным, не скрытым, не уникальным и изменяемым. Если не указан тип, то это строка текста. Тип **список_объектов** используется для выбора одного или множества объектов текущей модели, относящихся к указанным классам.Выбранные объекты будут представлены в виде списка идентификаторов с указанным в спецификации разделителем.

3. Секции входных и выходных контактов. *Название секции совпадает с идентификатором контакта*. Секция содержит следующие переменные:

а) **Name** – отображаемое имя контакта;

б) **Required** – логическое значение (0/1), определяющее, обязательно ли данный контакт должен иметь связь;

в) **Direction** – тип контакта, входной (**Input**) или выходной (**Output**);

г) **Type** – разрешение на проведение лишь одной (**Single**) или произвольного количества (**Many**) связей;

д) **Links** – переменная не пуста только для выходных контактов – это перечень допустимых к соединению контактов классов (как непосредственно указанных, так и их потомков). *После каждого элемента* перечня ставится «**;**» *и обязательный пробел*. Элемент перечня имеет вид:

элемент = идентификатор_класса «\» идентификатор_контакта

1.4.2. Порождающие классы

Каждый порождающий класс помимо специфицированных содержит стандартные поля и методы, формируемые системой порождения программ автоматически. Это поля

- **$ClassID** – идентификатор текущего класса;

- **$ID** – поле, содержащее идентификатор текущего объекта,

к которым добавляются все поля, представляющие специфицированные в файлах **Class.ini** свойства объектов такого класса. Имена таких полей совпадают с идентификаторами свойств.

В число методов (все они генерируются автоматически) входят

- стандартный конструктор, заполняющий идентификатор и все поля-свойства объекта;

- порождающий метод **Generate** с одним стандартным параметром **$Stage** (это идентификатор текущего события) и 2·K дополнительными параметрами, где K – общее число входных и выходных контактов класса (включая унаследованные). Каждому контакту с идентификатором **NAME** соответствуют два идущих подряд параметра – степень контакта (с именем **$powNAME**) и данные контакта (с именем **$NAME**). Данные выходных контактов передаются по ссылке, с модификатором **&**. Тело метода включает автоматически вставляемые небольшие пре- и пост-фрагменты, между которыми без каких-либо изменений вставляется код из соответствующего **script.php3**.

Таким образом, очевидно, что задача проектирования порождающих скриптов сводится, фактически, к написанию тел методов

Generate в файлах **script.php3**. При написании реализаций тел следует опираться на идентификатор события **$Stage**, на данные с входных контактов, переданные через параметры, а также, опционально, на запросы к XML-структуре[1], в которую отображается OCM, и на данные из общей базы данных класса «почтовый ящик», о которой будет сказано несколько слов далее[2]. Тело метода, на основе данной информации, может генерировать некий выходной фрагмент (с помощью **echo** или путем простого указания текста между тэгами **?>** и **<?**) и *должно* помещать выходные данные в параметры – выходные контакты. Также не исключена запись фрагмента в начало или конец текущей ленты **$Tape**, возможна работа с произвольными PHP-функциями.

При необходимости метод **Generate** может генерировать ошибку исполнения, которая (по завершении скрипта) будет показана в окне с результатами трансляции модели (будет выведен текст общего скрипта, в котором строка с указанной ошибкой будт выделена красным цветом). Для этого используется функция

MakeError($Class,$Text,$Line) – в первом параметре указывается класс ошибки, во втором – текст сообщения, в третьем – номер строки, которую следует подсветить (обычно передается значение **__LINE__**).

1.4.3. Контакты

Простейшим способом передачи данных между объектами являяется передача по контактам, которая реализуется через параметры методов **Generate**. Сложность такого подхода состоит в том, что данная передача возможна только в направлениях, определяемых связями модели, и, если необходимо передать данные между объектами, кото-

[1] Доступна через глобальный объект **$XML** класса **XPathEngine** проекта **Php.XPath 3.5**, см., например, http://sourceforge.net/projects/phpxpath/. Данный объект создается и инициализируется автоматически.

[2] Кроме того, не исключено и использование обычных глобальных переменных, хотя такая практика не поощряется.

рые не соединены непосредственно, то о такой передаче должен позаботиться не только отправитель, но и промежуточные объекты на пути следования данных.

Следует особо упомянуть о *структуре данных, передаваемых через контакты (то есть через параметры метода **Generate**)*. Каждая такая структура **M** всегда является двумерным ассоциативным кортежем

M[“имя_параметра”][номер_связи_на_контакте],

где в качестве имени параметра (ячейки) может использоваться произвольный идентификатор, а номер связи в случае, если связь только одна, будет нулевым. Отметим, что существуют предопределенные элементы:

- **M[“_ClassID”][k]** – идентификатор класса объекта, присоединенного по связи номер **k**;

- **M[“_LinkID”][k]** – идентификатор контакта объекта, из которого исходит связь номер **k**;

- **M[“_ID”][k]** – идентификатор объекта, присоединенного по связи номер **k**.

Эти предопределенные параметры облегчают анализ структуры связей в модели, давая объектам информацию о связанных с ними объектами.

Читать данные из структур вида **M** можно таким же образом, как и из обычного массива. Помещать же данные **B** в выходной контакт **A** рекомендуется с применением стандартной функции

cortege_push(A[“имя_параметра”], B).

1.4.4. События

Всегда существуют четыре предопределенных события: **stResource**, **stInit**, **stCall** и **stDone**, которые происходят именно в таком порядке. Любой порождающий скрипт всегда может спланировать новые события с помощью функций:

CreateEventBefore($Before,$Line) – создает новое событие перед событием **$Before**, в параметр **$Line** передается номер строки **__LINE__** (необходим при генерации сообщений об ошибках);

CreateEventAfter($After,$Line) – создает новое событие после события **$After**, в параметр **$Line** передается номер строки **__LINE__** (необходим при генерации сообщений об ошибках).

Упомянем еще две полезные функции:

GetLastEvent($Ev) – возвращает идентификатор события из массива событий **$Ev**, которое произойдет последним из них;

NextEvent() – возвращает идентификатор ближайшего следующего события.

1.4.5. «Почтовый ящик»

«Почтовый ящик» – глобальный ассоциативный кортеж – контейнер, элементами которого могут быть структуры произвольной сложности. Обычно используется для быстрой передачи данных между объектами модели без привлечения контактов (в том числе при обработке различных событий). Для работы с ним используются следующие функции:

PutMail($Name,$Val) – помещает значение **$Val** в очередь значений ячейки с именем **$Name**;

GetNextMail($Name) – извлекает следующее значение из очереди значений ячейки с именем **$Name**, при этом значение удаляется из очереди;

ReadNextMail($ID,$Name) – читает (не удаляя) следующее значение из очереди значений ячейки с именем **$Name**, при этом используется внутренний указатель очереди с произвольным именем **$ID** (если такого указателя на момент обращения не существует, то он будет создан). Данная функция позволяет, используя различные **$ID**, читать значения из очереди многократно.

Последние две функции в случае, если очередь значений закончилась, возвращают пустую строку.

1.4.6. Лента системы

Лента – эволюционный оператор ОСМ. Она представляет собой строку, включающую операторы на языке PHP, которые исполняются в конце обработки каждого события, причем лента автоматически не очищается. Понятие ленты было введено для совместимости с концепцией расширенных машин Тьюринга (РМТ) [9].

Лента может использоваться для динамического формирования и исполнения PHP-кода, что может повысить производительность в ряде случаев, например, для генерации и быстрого исполнения фрагментов, вычисляющих отклик нейронной сети, заданной в виде ОСМ, и производящих обратное распространение ошибки [9].

Для работы с ней могут использоваться следующие функции:

AppendTapeBegin($Text) – добавляет строку **$Text** в начало ленты;

AppendTapeEnd($Text) – добавляет строку **$Text** в конец ленты;

ClearTape() – очищает ленту;

GetTape() – возвращает содержимое ленты.

1.5. Практические аспекты реализации решающих методов

Как уже отмечалось выше, основной целью этапа трансформирующей интерпретации, выполняемой объектами решающих (дедуктивных) классов, является анализ структуры модели с последующим синтезом элементов плана решения или постановки задачи, которые оформляются путем ввода в модель новых объектов и модификации

параметров уже существующих объектов. При этом исходные объекты решающих классов из модели удаляются. Данные задачи имеют преимущественно логический характер и выполняются с помощью инициализирующих, решающих и вспомогательных предикатов дедуктивных классов, написанных на языке **GNU Prolog**.

Этап трансформирующей интерпретации проходит в две стадии. На первой стадии происходит цепочка вызовов инициализирующих предикатов, запускаемых в порядке, определяемом связями модели в соответствии с идеологией сетевого графика работ. На второй стадии эквивалентным образом вызываются решающие предикаты. Если класс не содержит описаний как инициализирующих так и решающих предикатов, то считается, что он *не является решающим*. Если же отсутствует описание лишь одного из указанных предикатов, то считается, что он существует, пуст и является истинным.

Инициализирующий предикат класса оформляются в виде скриптового файла **init.pl**, находящегося в каталоге соответствующего дедуктивного класса. Данный файл содержит обязательный инициализирующий предикат

init(ID, параметры_входы, параметры_выходы),

а также произвольное множество вспомогательных предикатов. **Параметры_входы** при вызове предиката являются связанными и содержат значения, переданные текущему решающему объекту предыдущими (по связям) решающими объектами. Их количество равняется количеству всех входов (включая унаследованные). **Параметры_выходы** должны получать значения в результате вызова данного предиката, их количество равняется количеству всех выходов (включая унаследованные). Порядок следования этих параметров-контактов в заголовке предиката определяется порядком перечисления соответствующих им секций в файлах **Class.ini**, причем сначала идут контакты родительских классов.

Если к какому-либо входу объекта связи не подключены, то в соответствующий параметр передается строка **‘null’**. Если подключена строго одна связь, то во входной параметр передается значение,

помещенное в соответствующий выходной параметр объектом, из которого исходит связь. Если подключено более одной связи, то в параметр передается список значений, помещенных в соответствующие выходные параметры-контакты объектами-источниками связей.

Решающий предикат класса оформляется в виде скриптового файла **solve.pl**, находящегося в каталоге соответствующего дедуктивного класса. Данный файл содержит обязательный решающий предикат

solve(ID, параметры_входы, параметры_выходы),

а также произвольное множество вспомогательных предикатов. Состав параметров и правила передачи данных в параметры-контакты в точности такие же, как и для инициализирующих предикатов.

1.5.1. Работа с моделью

Текущая модель, с точки зрения вышеупомянутых предикатов, представляется в виде динамической базы фактов, соответственно, все операции с моделью опосредуются стандартными операциями унификации, вставки и удаления этих фактов из базы. Опишем некоторые стандартные факты:

pgen_system(язык_конкретизации) – определяет язык конкретизации модели;

elements(число_элементов) – определяет число элементов модели;

element(идентификатор, идентификатор_класса, идентификатор_родительского_элемента, постоянство_идентификатора) – определяет отдельный элемент;

parameters(идентификатор, количество_параметров) – определяет число параметров элемента;

parameter(идентификатор, идентификатор_параметра, размер_отступа, значение_параметра) – определяет значения параметров элемента;

show(идентификатор, показыв_класс, показыв_идентиф, показыв_иконку_класса) – определяет флаги отображения элемента;

position(идентификатор, абсцисса_лев_верх_точки, ордината_лев_верх_точки) – определяет позицию элемента;

internal_inputs(идентификатор, число_входов) – определяет число входов элемента;

i_contact(идентификатор, идентификатор_контакта) -- определяет один из входных контактов элемента;

internal_outputs(идентификатор, число_выходов) – определяет число выходов элемента;

o_contact(идентификатор, идентификатор_контакта) -- определяет один из выходных контактов элемента;

i_link(идентификатор_входного_объекта, идентификатор_входного_контакта, идентификатор_выходного_объекта, идентификатор_выходного_контакта, признак_информ_связи) – определяет исходящую связь;

o_link(идентификатор_выходного_объекта, идентификатор_выходного_контакта, идентификатор_входного_объекта, идентификатор_входного_контакта, цвет_связи, признак_информ_связи, число_виз_точек_связи, строка_координат_точек) – определяет входящую связь.

Существуют три высокоуровневых предиката, предназначенных для вставки и удаления элементов (объектов), а также для вставки связей (удаление связей происходит автоматически при вызове предиката удаления элемента):

insert_element(идентификатор, идентификатор_класса, показыв_класс, показыв_идентиф, показыв_иконку_класса, абсцисса_лев_верх_точки, ордината_лев_верх_точки, список_идентификаторов_парам-ов, список_значений_парам-ов, список_идент_вход_контактов, список_идент_выход_контактов) – вставка элемента (объекта) в модель;

delete_element(идентификатор) – удаление элемента (и всех его связей) с указанным идентификатором;

insert_link(идентификатор_выходного_объекта, идентификатор_выходного_контакта, идентификатор_входного_объекта, идентификатор_входного_контакта, цвет) – создание основной связи между двумя объектами.

1.6. Некоторые приложения

Изложенная в данной главе схема порождения реализована автором [11] в программной системе PGEN++, которая, в настоящее время, содержит иерархии классов для решения ряда учебных и практических задач, например:

а) генерации настроечного кода для системы моделирования распространения загрязнений AirEcology-P по блочному описанию моделируемой задачи;

б) генерации, обучения и упрощения глубоких нейронных сетей прямого распространения, интерполирующих заданные наборы данных;

в) генерации простых учебных программ, решающих задачи элементарной обработки векторных данных (поиска минимума, максимума, среднего арифметического).

Далее приведен несложный пример порождающего PHP-метода для класса, осуществляющего генерацию кода для вывода на экран переменной при решении простых учебных задач обработки векторных данных.

```
<? if ($Stage == stInit) {
      $argumentID = $Arg["ID"][0];
      if ($Arg["_ClassID"][0] == "clsSimpleVector") {
         $argumentSize = $Arg["Size"][0];
?>
  printf("<? echo $argumentID; ?> = [");
  for (i = 0; i < <? echo $argumentSize; ?>; i++)
<?       echo "    printf(\"%lf \", $argumentID" . "[i]);\n";
?>
```

```
  printf("]\n");
<?    } else
         echo "    printf(\"$argumentID = %lf\\n\", $argumen-
tID);\n";
   }
?>
```

1.7. Представление алгоритмов и планов решения задач в виде ОСМ

Как было показано выше, ОСМ, фактически, способна представлять высокоуровневое описание задачи. По этому описанию сначала может дедуктивно выводиться детализированный план решения, а потом порождаться решающая поставленную задачу программа. При этом не налагается каких-либо существенных ограничений на вид исходного описания – это может быть как *формальная постановка задачи* исключительно в терминах объектов предметной области и связей между ними [11], так и некий *план решения задачи*, включающий не только объекты, но и описания производимых над ними трансформаций [9]. В частном случае, это может быть даже блочное представление (диаграмма) алгоритма решения задачи на некотором языке, являющееся, например, блок-схемой (схожий подход описан в [10, 14]). Еще одним частным случаем исходного описания может быть текст (например, естественно-языковая постановка задачи), если он имеет однозначное отображение в ОСМ.

Отсюда следуют, по меньшей мере, *два важных вывода*:

1. Если имеется естественно-языковое описание постановки задачи, то, имея некоторый индуцирующий компонент, распознающий и преобразующий эту постановку в ОСМ, можно построить *естественно-языковой интерфейс* к системе порождения программ.

2. Если имеется некоторая (уже порожденная системой PGEN++) программа, то, при наличии некоего индуцирующего компонента, последовательно распознающего ее в формате линейной

«синтаксической» ОСМ с последующей трансформацией в полноценную сетевую «смысловую» модель, можно проверить на тождественность полученную и исходную порождающую модели, и, тем самым, фактически, верифицировать порождающие скрипты.

Построив адекватный индуцирующе-трансформирующий компонент системы порождения программ PGEN++, мы создадим предпосылки для решения не только задачи порождения программ по естественно-языковым описаниям, но и, косвенно, задачи верификации порождающих скриптов.

Данная проблема будет подробнее рассмотрена далее.

Выводы к первой главе

В данной главе изложены базовые теоретические и некоторые практические сведения об основных структурах данных и связанных с ними порождающих механизмах системы PGEN++. Дано понятие объектно-событийной модели, описаны ее структура и правила интерпретации, позволяющие порождать программу по исходной постановке задачи, данной в произвольной форме (при условии, что она однозначно отображается в порождающую ОСМ). В частности, таким путем потенциально может быть решена задача порождения программ по естественно-языковым постановкам, содержащим описание задачи и/или план/алгоритм ее решения.

Глава 2. Индукция первичной ОСМ алгоритма/плана по программе/тексту

Целью данной главы является рассмотрение вопросов, связанных с решением базовой проблемы распознавания постановки задачи с генерацией первичной смысловой модели.

Для реализации данной цели поставим следующие задачи:

а) предложить общую схему восстановления линейных ОСМ (первичных описаний алгоритма/плана) по неструктурированному (текст на естественном языке) или структурированному тексту (в частном случае – по программе);

б) предложить схему оперативной логико-алгоритмической обработки первичных результатов разбора входного текста (полученных как с привлечением, так и без привлечения грамматического разбора) с заполнением полей объектов генерируемой ОСМ;

в) сформулировать синтаксические средства описания распознающей части индуцирующих скриптов;

г) рассмотреть некоторые теоретические вопросы, связанные с проблемами реализуемости поставленных задач и алгоритмической полноты используемых средств.

2.1. Общий подход к индукции первичной ОСМ

Решение задачи индукции ОСМ в наиболее общем случае может рассматриваться как вывод ОСМ из знаний, заключенных в произвольном структурированном (например, программе) или неструктурированном тексте (в частности, на естественном языке).

Предположим, что процесс индукции в общем виде может быть описан распознающей ОСМ, отличающейся от «обычной» порождающей ОСМ тем, что вместо объектов используются непосредственно классы, для которых с помощью сети определена последователь-

ность применения неких индуцирующих методов. В ходе работы методов (на первом этапе) генерируются объекты-элементы первичной ОСМ, которая изначально представляет собой цепочку, построенную в порядке «чтения» анализируемого текста (программы или, например, описания задачи на естественном языке). Данный этап будет подробно рассмотрен в текущей главе. На втором этапе работы (будет рассмотрен в следующей главе) применяются логические схемы (прямого или обратного логического вывода) индуцирующих методов, которые вводят необходимые дополнительные объекты, устраняют временные объекты и необходимым образом реорганизуют систему связей между полученными объектами, окончательно формируя выходную смысловую модель-ОСМ.

2.1.1. Распознающие ОСМ

Итак, индуцирующая (распознающая) ОСМ представляет собой сеть классов, которые, вообще говоря, являются расширениями обычных порождающих классов, содержащими индуцирующие методы

Индуцирующая ОСМ является графом (R, E), где R — множество узлов (индуцирующих классов $R \subseteq C$, где C — множество всех классов), E — множество дуг (связей между классами). Индуцирующий класс S также является пятеркой (N_c, I, O, F, M), где N_c — идентификатор (имя) класса, I и O — наборы входных и выходных контактов (определяются функциями $I = in(S)$, $O = out(S)$), F — поля, $M = M_G \cup M_I$, M_G — порождающие методы, M_I — индуцирующий метод.

В совокупности, индуцирующие методы должны выполнять *две основные задачи*: а) разбор предложенного им на вход текста с построением его фактологического описания в виде первичной линейной ОСМ, и б) вывод результирующей ОСМ, представляющей совокупность знаний об анализируемом тексте, по данному описанию.

Первая задача может быть решена с помощью каких-либо модифицированных средств синтаксико-семантического разбора и анализа, которые, *для ускорения работы*, должны самостоятельно обрабатывать большинство типичных и часть атипичных случаев совпадения с образцами, а также отсеивать ложные (противоречащие грамматике и/или семантике) срабатывания еще на уровне шаблона. Например, при разработке естественно-языкового интерфейса необходимо сразу отсеивать заведомо неверные сочетания языковых элементов. Здесь наиболее перспективным вариантом представляется некоторое *расширение синтаксиса и функционала классических регулярных выражений*, которые можно было бы дополнить следующими возможностями:

1. Опциональным *скрытым трансформационным слоем*, преобразующим «сырые» данные входного текста в некоторое частичное, но четко структурированное выходное описание, что может потребоваться, например, для упрощения обработки сложных входных естественно-языковых текстов. Наиболее целесообразна реализация такого слоя в виде подключаемых внешних библиотек, реализующих функционал слоя. Адекватным вариантом представляется организация стандартного интерфейса с таким слоем на базе XML/XPath-технологий. Простым примером трансформационного слоя может быть слой грамматического разбора. Заметим однако, что в случае входных текстов простой структуры, вполне достаточным может оказаться использование исключительно возможностей регулярных выражений.

2. Некоторыми простыми *дополнительными элементами логического программирования*, например, со следующими классами предикатов: а) расширенными средствами простого и нечеткого сравнения разобранных элементов с образцами по таблице/таблицам, б) классификации таких элементов (или обратной задачи восстановления недостающих параметров элементов по известному классу) нейронной сетью, в) выполнения XPath-подобной обработки текстовых XML-фрагментов (в том числе фрагментов с результатами работы

слоя грамматического разбора, что и обеспечивает упомянутый выше интерфейс с данным слоем), г) выполнения неких простых алгоритмических фрагментов, описанных на некотором вспомогательном микроязыке (например, весьма интересным представляется применение в качестве такового XPath-подобного языка, поскольку применение XPath-технологий уже обозначено в предыдущем пункте), д) произвольными предикатами из динамически подключаемых библиотек.

Вторая задача априорно близка задаче дедуктивного логического вывода (прямого или обратного), поскольку весьма вероятна необходимость решения достаточно интеллектуальных задач анализа первичных знаний, извлеченных и, возможно, предобработанных с помощью неких шаблонов в ходе решения первой задачи.

Таким образом, целесообразно представить индуцирующий (распознающий) метод M_I как тройку вида $M_I = (M_L, M_{LD}, T)$, где M_L – фрагмент кода на языке логического программирования (логический скрипт обратного вывода), осуществляющий обработку первичных знаний по схеме обратного логического вывода, M_{LD} – фрагмент кода на некотором специальном языке (логический скрипт прямого вывода), осуществляющий обработку первичных знаний по схеме прямого логического вывода, T – шаблон-группа из полученных индуктивным путем *модифицированных регулярных выражений* (результатов экспертного обобщения частных случаев входного текста). Именно шаблоны позволяют решить первую задачу – построение первичной ОСМ, включающей извлеченные из постановки задачи факты. Скрипты, соответственно, ориентируются на решение второй задачи.

Интерпретация распознающей ОСМ выполняется, в целом, по схеме «содержательной интерпретации», заимствованной из обычной ОСМ. В данном контексте было бы уместнее назвать интерпретацию классово-сетевой. При этом также присутствует календарь событий (с одним начальным событием, которое в простейшем частном случае может быть и единственным), который при необходимости может пополняться в ходе интерпретации событий индуцирующими методами

(в явной или неявной форме), а также некоторая общая база данных[1] (аналог «почтового ящика»), через которую, преимущественно, и осуществляется взаимодействие индуцирующих методов. Необходимо заметить, что логические скрипты обратного вывода M_L исполняются непосредственно сразу после отработки шаблона T индуцирующего метода, тогда как скрипты прямого логического вывода M_{LD} в ходе основной интерпретации собираются в единый пакет M_{DIR}, который на заключительном этапе запускается на *специализированной машине прямого вывода*. Соответственно, возможны три варианта (вырожденный случай без скриптов не рассматривается):

а) $\forall M_I : M_L \neq \varnothing, M_{LD} = \varnothing$ – режим интерпретации с исключительно обратным выводом;

б) $\forall M_I : M_L = \varnothing, M_{LD} \neq \varnothing$ – режим интерпретации с исключительно прямым выводом;

в) $\exists M_I : M_L \neq \varnothing, M_{LD} \neq \varnothing$ – режим интерпретации со смешанным выводом;

Определенное подмножество построенных элементов полученной общей базы, в конечном итоге, и определяет выходную ОСМ. Отметим, что порядок запуска методов классов также определяется по схеме сетевого графика работ, как и в порождающей ОСМ.

Вышеуказанный общий алгоритм интерпретации на практике сводится к двум более простым стандартным схемам обхода (индукции), которые будут представлены позднее, в специальных подразделах.

[1] В случае обратного логического вывода это внутренняя база фактов машины вывода GNU Prolog, а в случае прямого логического вывода, основанного на XPath-подобных заключениях, это содержимое текущего XML-документа, представляющего текущий вариант входо-выходной смысловой ОСМ.

2.1.2. Схемы индукции

Выше был описан общий алгоритм интерпретации, характерный для распознающих ОСМ, причем было отмечено, что на практике вполне достаточно применения двух редуцированных схем. В этом небольшом разделе данные схемы будут подробно описаны.

2.1.2.1. Индукция первичных моделей по произвольному тексту

В произвольном случае текст может быть как неструктурированным, так и слабо структурированным. При этом элементы текста, отражающие некоторый набор знаний, в нем содержащихся, могут присутствовать почти в произвольном порядке[1] и описываться контекстно-зависимыми грамматиками. Кроме того, весьма вероятно наличие текстовых элементов, не несущих существенной содержательной информации, которые при разборе должны быть пропущены. Отсюда следует затрудненность или даже невозможность применения схем индукции, подобных автоматным. Скорее всего, здесь будет более оправдана *сканирующая или «поисковая» технология*, согласно которой распознающая ОСМ (сеть классов) будет, фактически, интерпретироваться в рамках *единственного метасобытия*, когда порядок вызова индуцирующих методов классов будет определяться исключительно структурой связей графа модели. При этом *индуцирующий метод каждого класса будет вызываться многократно, в цикле (который технически вполне может быть организован с помощью обычной для ОСМ событийной петли)*, сканируя входной текст от начала к концу и вызывая логический обрабатывающий скрипт M_L *при обнаружении каждого очередного совпадения* шаблона T с фрагментом текста (технология, схожая с работой в текстовом редакторе, когда запускается процесс поиска некоторого образца и для каждого обна-

[1] Особенно это характерно для текстов на естественном языке, в котором фразы, относящиеся к одному понятию, иногда могут располагаться в произвольном порядке. То же может быть справедливо для некоторых компонентов одной фразы.

руженного вхождения выполняется некоторое заранее определенное множество действий). Видимо, при каждом совпадении будет генерироваться некоторый объект выходной ОСМ, место которого в такой модели целесообразно определить, считая, что *выходные объекты следуют в том же порядке, в котором порождающие их образцы следуют во входном тексте*. Содержание *полей выходного объекта* будет определяться значениями параметров (переменных) шаблона T, полученными в результате унификации T с фрагментом/фрагментами входного текста.

По окончании обработки метасобытия будет вызван воспол-няющий/проверяющий полученную модель псевдоскрипт M_{DIR}, являющийся конкатенацией скриптов M_{LD}, относящихся к рассматриваемому множеству индуцирующих классов, и интерпретируемый специальной машиной прямого логического вывода.

Необходимо отметить, что целесообразно предусмотреть и *частный редуцированный случай переработки элементов входного текста по типу «поиск-замена»*, когда скрипт M_L, фактически лишь заменяет найденный фрагмент иным фрагментом (константным, или построенным по каким-либо простым синтаксическим правилам). Такой случай может иметь место как для неструктурированных, так и структурированных текстов, например, в случае простейшей прямой переработки комментариев в С-программах, путем их замены пустыми строками. Данный редуцированный случай рассмотрен, в частности, в пункте 3.6.4.1.

Следует учесть, что представленная в данном пункте схема индукции, скорее всего, является достаточно затратной по времени исполнения. Поэтому, для структурированных текстов (в частности, программ) целесообразно применение более быстрых алгоритмов, не требующих многократно перезапускаемого комбинационного полнотекстового поиска.

2.1.2.2. Индукция первичных моделей по программе

В отличие от предыдущего случая, здесь любой элемент входного текста (программы), во-первых, обычно имеет значение для понимания этого текста, во-вторых, порядок следования таких элементов чаще всего строго определен и детерминируется грамматикой языка/языков программы.

Можно предположить, что допустима последовательная обработка входного текста, когда для каждого его очередного фрагмента F заново запускается сеть классов распознающей ОСМ (фактически реализуется основной событийный цикл модельной интерпретации, завершающийся при достижении конца входного текста). В ходе работы сети последовательно (в соответствии с идеологией сетевого графика работ) проверяется соответствие текущего фрагмента каждому очередному шаблону T_i из очередного каскада классов W, $i \in W$. Если некоторый j-й класс определяет соответствие своему шаблону T_j, то

а) вызывается логический скрипт $(M_L)_j$, выполняющий необходимый процессинг разобранного фрагмента F по T_j с формированием очередного элемента выходной цепочечной ОСМ (элементы будут следовать в том же порядке, что и соответствующие им фрагменты входного текста);

б) обработка текущего события завершается (прекращается текущая *сетевая* интерпретация модели);

в) выделяется следующий фрагмент входного текста (если он есть), для которого происходит следующее событие с новой интерпретацией сети.

Как и в случае индукции по произвольному тексту, по окончании основного цикла будет вызван производящий заключительную обработку и проверку псевдоскрипт M_{DIR} (конкатенация скриптов M_{LD}, относящихся к рассматриваемому множеству индуцирующих классов), интерпретируемый специальной машиной прямого логического вывода.

Такая схема, несомненно, менее ресурсоемка и более детерминирована. При этом наличие строгой формальной входной грамматики позволяет предположить, что возможно построение такой сети классов, которая не будет противоречить данной грамматике и позволит выделить все требуемые для понимания входного текста элементы.

Относительно порядка следования классов в распознающей ОСМ, индуцирующей, например, С-программу, следует дополнительно отметить, что целесообразна схема последовательного применения «от менее общего к более общему». В частности, такие наиболее конкретные классы как «заголовок цикла while» должны предшествовать «произвольному оператору», который, в свою очередь, должен предшествовать «программе в целом» (шаблон Т такого класса, вероятнее всего, будет проверять лишь факт достижения конца входного текста). Кроме того, последовательность классов должна быть такой, чтобы их шаблоны однозначно и корректно унифицировались с соответствующими текстовыми фрагментами.

2.1.3. Регулярно-логические выражения

Как уже было отмечено выше, шаблоны, входящие в состав индуцирующих скриптов, должны содержать группы унифицирующих выражений, которые:

а) являются контекстно-зависимыми;

б) имеют явные элементы-переменные, получающие значение при успешной унификации выражения с очередным фрагментом текста.

Дополнительно отметим, что такие выражения должны быть достаточно просты и понятны программирующему пользователю системы (разработчику индуцирующих скриптов). Эти обстоятельства достаточно определенно указывают на регулярные выражения, в полной мере соответствующие заявленным требованиям. Более того, идея

их применения хорошо укладывается в обе предложенные выше схемы индукции, как сканирующую (поисковую) для произвольного текста, так и последовательную, наиболее адекватную для структурированных текстов с четко формализованной грамматикой.

Выше уже отмечалось, что шаблоны, в целях повышения качества предварительного выделения искомых фрагментов и общего повышения скорости работы, должны содержать некие простые элементы логического программирования, например, со следующими классами предикатов:

а) с расширенными средствами простого и нечеткого сравнения разобранных элементов с образцами по таблице/таблицам,

б) со средствами классификации таких элементов (или обратной задачи восстановления недостающих параметров элементов по известному классу) нейронной сетью,

в) выполняющих XPath-подобную обработку текстовых XML-фрагментов (в том числе фрагментов с результатами работы упоминавшегося ранее трансформирующего слоя [грамматического разбора], что существенно упрощает задачу распознавания смысла входных текстов на естественном языке),

г) выполняющих некие простые алгоритмические фрагменты, описанные на некотором вспомогательном микроязыке (например, весьма интересным представляется применение в качестве такового XPath-подобного языка, поскольку применение XPath-технологий уже обозначено в предыдущем пункте),

д) произвольные предикаты из динамически подключаемых библиотек.

Такой набор предикатов существенно облегчает синтаксико-семантический разбор и повышает его интеллектуальность, обеспечивая, в частности, выявление грамматических связей, проверку повторяемости, проверку синонимов по таблице, проверку схожести (почти исключающую влияние небольших синтаксических ошибок-описок на результат), четкую или нечеткую классификацию, восстановление

классифицирующих признаков по известному классу, а также некоторые виды дополнительной простой алгоритмической обработки.

Классические регулярные perl-выражения [26] уже, фактически, содержат логическую схему последовательной унификации с откатами, также имеется возможность встраивания исполняемого/проверяющего кода. Указанный встраиваемый код, теоретически, позволил бы реализовать большинство требуемых возможностей, однако такой подход с «прямым программированием» не вполне удобен, поскольку встраиваемый код не является универсальным (он соответствует основному используемому в конкретной реализации регулярных выражений языку программирования), что резко снижает его ценность.

Соответственно, *актуальна разработка дополнительных средств, расширяющих классические регулярные выражения* вышеперечисленными элементами логического (и простейшего XPath-алгоритмического) программирования в составе некоторого дополнительного платформонезависимого логического микроязыка, предикаты которого укладываются в концепцию работы с откатами (по крайней мере в части работы с таблицами) и имеют стандартный интерфейс с трансформирующим слоем [грамматического разбора], а множество таких предикатов является расширяемым за счет использования динамически подгружаемых библиотек. Получаемые при этом регулярные выражения назовем *регулярно-логическими выражениями*. Дополнительно отметим, что такой подход позволяет обрабатывать вводимые логические микровыражения с достаточно высокой скоростью за счет *естественного распараллеливания их обработки*. Данный вопрос будет кратко рассмотрен в отдельном пункте.

2.1.3.1. Элементарные переменные

В классических регулярных выражениях основным элементом, имеющим видимое извне значение, является пара круглых скобок, которая может иметь имя. Видимо, имеет смысл сохранить эту традицию при выделении переменных, но, поскольку их семантика не-

сколько отлична от классической, целесообразно ввести новый синтаксис именования переменной (пары скобок), подчеркнув тем самым специфику их применения в логических микровыражениях. Тогда элементарную *переменную* регулярно-логического выражения обозначим так:

переменная = «(» фрагмент_шаблона «)**->{**» идентификатор «**}**»

Идентификатор не может начинаться с цифры, это позволяет не путать выделение переменной и применение квантификатора {MIN,MAX}. Как и любой иной значащий элемент выражения, *переменная может иметь квантификатор*, который указывается непосредственно после «**}**».

Замечание о полиморфизме переменных. Основным существенным семантическим отличием предложенных переменных от классических является то, что при неоднократной успешной унификации соответствующего фрагмента шаблона (например, в выражении «**(\d+\s+)->{P}+**» переменная P может последовательно получить серию значений) *все* получаемые значения сохраняются. Учитывая, что с синтаксической точки зрения переменные могут быть вложенными (например, в выражении «**((\d+)->{Num}\s+)->{P}+**»), имеем *дерево переменных/значений регулярно-логического выражения*. Корнем дерева является выражение в целом. Далее следуют поочередно уровни переменных и уровни значений этих переменных (таким образом любая переменная A может иметь множество подчиненных вершин-значений, каждое из которых имеет множество подчиненных вершин-переменных, которые синтаксически вложены в A.

Как и в обычных регулярных выражениях, имеет смысл определить две простые синтаксические конструкции для сравнения значения скобочного выражения с ранее полученным значением переменной на равенство и на неравенство:

проверка_равенства = «(» фрагмент_шаблона
«)**==>{**» полный_идентификатор «**}**»
проверка_неравенства = «(» фрагмент_шаблона
«)**!=>{**» полный_идентификатор «**}**»

где полный_идентификатор является либо *ссылкой* на *переменную текущего выражения*, либо (в синтаксисе с точкой) *ссылкой* на *переменную иного выражения*:

полный_идентификатор = идентификатор | имя_выражения «**.**» идентификатор

Результаты сравнения, соответственно, определяют успешность унификации приведенных выше выражений.

2.1.3.2. Переменные трансформирующего слоя

Особо выделим трансформирующие переменные, которые для получения своего значения задействуют трансформирующий слой. Закономерен вопрос – почему такие переменные выделены особым образом (казалось бы, можно использовать обычные элементарные переменные и обрабатывать полученные ими значения в трансформирующем слое, который был бы оформлен в виде обычных внешних библиотечных предикатов)? Ответ состоит в том, что при работе с трансформационным слоем важен не только конечный результат, но и промежуточные итоги его работы, которые являются важными метриками входной задачи, используемыми машиной прямого логического вывода для интерполирования вероятностей перехода между правилами по параметрам схожести задач. Поэтому, взаимодействие с таким слоем должно быть выделено особым, понятным системе PGEN++ способом.

Трансформирующие переменные (Т-переменные) синтаксически оформляются следующим образом:

Т-переменная = «**(**» фрагмент_шаблона «**)->{**»
[имя_трансформ_библиотеки «.» имя_режима] «**{**» идентификатор_переменной «**}**» «**}**»
идентификатор_переменной = идентификатор
имя_трансформ_библиотеки = идентификатор
имя_режима = идентификатор

Такая запись обозначает, что выделенное фрагментом шаблона подвыражение поступает на разбор/анализ в трансформирующий

слой, определенный либо указанием имени трансформирующей DLL-библиотеки и режима трансформации, либо библиотекой по умолчанию, определенной в настроечном файле поддерева индуцирующих классов. Результат трансформации (по умолчанию считаем, что это фрагмент некоего результирующего XML-файла, из которого удален корневой тэг) попадает в переменную с указанным идентификатором.

В настоящее время система работает с единственным видом трансформирующего слоя – со слоем грамматического разбора. Это трансформирующая библиотека Grammar.dll с двумя режимами работы (en – английский язык, ru – русский язык), которая, в свою очередь, использует стандартную библиотеку выявления грамматических связей в предложении Link Grammar 5.3.0 [23].

Как и было специфицировано выше, выходная информация библиотеки Link Grammar (набор выявленных грамматических связей между словами) представляется в виде фрагмента виртуального XML-документа, в связи с чем, в набор стандартных микропредикатов регулярно-логических выражений потребовалось ввести специальные средства для работы с такими XML-документами, хранимыми в T-переменных.

2.1.3.3. Логические микровыражения

Логическое микровыражение также, в конечном итоге, привязывается к паре скобок и выполняет *цепочку предикатов*, оперирующих с этим значением. Внутри микровыражения текущее значение определяется так:

текущее_значение = «**$**»

а само микровыражение представляется следующим образом:

микровыражение = добавление | удаление | обучение | проверка

добавление = «(» фрагмент_шаблона «)**+=>{**» цепочка_предикатов «**}**»

удаление = «(» фрагмент_шаблона «)**-=>{**» цепочка_предикатов «**}**»

обучение = «(» фрагмент_шаблона «)***=>{**» цепочка_предикатов «**}**»

проверка = «(» фрагмент_шаблона «)**?=>{**» цепочка_проверок «**}**»

Результат унификации микровыражения существенно зависит от его типа: выражения классов «добавление», «удаление» и «обучение» успешны всегда (их элементами, фактически, являются параметризованные упоминания предикатов, представляющие выражения, которые следует добавить в базу данных или удалить из нее, или которые описывают план создания/обучения нейронной сети [2, 15]), а успешность выражения класса «проверка» определяется истинностью цепочки предикатов: унификация успешна тогда и только тогда, когда истинны все члены цепочки проверок. При этом *цепочка проверок вычисляется строго в порядке «слева-направо»*, завершаясь при первой же неудачной проверке или при достижении конца цепочки.

Далее:

цепочка_предикатов = элемент_цепи {«**,**» элемент_цепи}

элемент_цепи = параллельный_старт | последовательный_старт | предикат

параллельный_старт = «**parallel**» [«**(“analyze”)**»]

последовательный_старт = «**sequential**»

цепочка_проверок = элемент_проверки {«**,**» элемент_проверки}

элемент_проверки = параллельный_старт | последовательный_старт | [отрицание] предикат

отрицание = «**!**»

Для определения истинности некоего элемента проверки вычисляется истинность соответствующего предиката (псевдопредикаты **parallel** и **sequential** истинны всегда), на которую может (при наличии префикса «**!**») накладываться отрицание.

2.1.3.4. Замечания о параллельной обработке цепочек предикатов

Псевдопредикат параллельного старта **parallel** запускает цепь следующих после него предикатов (до конца или до ближайшего псевдопредиката последовательного старта) *в максимально возможном параллельном режиме с учетом наличия зависимостей между предикатами по данным*. Если данный псевдопредикат имеет параметр **«analyze»**, то включается *интеллектуальный режим исполнения*,

при котором распараллеливание включается тогда и только тогда, когда *внутренний предиктор* (выполняющий предсказание времен исполнения в параллельном [t_1] и последовательном [t_N] режимах, оперируя линейным предиктором для t_1, линейными соотношениями по типу закона Амдала [1] для t_N и производными от них квадратичными выражениями, полученными в предположении линейной связи доли параллельной работы α с t_1) сочтет его оправданным. Полное рассмотрение данной схемы выходит за рамки настоящей работы, заметим лишь, что все предикторы проходят периодический сброс для повышения «актуальности» предсказаний, а само по себе интеллектуальное распараллеливание достаточно стабильно дает результаты, которые обычно имеют компромиссный характер: чаще лучше, чем для строго последовательного режима и могут быть (в зависимости от конкретных условий) как лучше, так и хуже строго параллельного решения. Это можно объяснить тем, что сбор данных для предикторов и их пересчет являются операциями, вносящими дополнительные накладные расходы, не идеальны и результаты предсказания.

Псевдопредикат последовательного старта **sequential** запускает цепь следующих после него предикатов (до конца или до ближайшего псевдопредиката параллельного старта) *в последовательном режиме*.

2.1.3.5. «Быстрые» предикаты

Определим сначала *синтаксис предиката в микровыражении*:

предикат = [имя_модуля «**.**»] имя_предиката [«(» список_параметров «)»]

имя_предиката = идентификатор

список_параметров = параметр {«**,**» параметр}

параметр = входной_параметр | произвольное_значение | параметр_по_ссылке

входной_параметр = константа | текущее_значение | полный_идентификатор

константа = целое_число | строка

текущее_значение = «**$**»

строка = «”» символы «”»

произвольное_значение = «_»

параметр_по_ссылке = «**$**» идентификатор

Приведенный выше синтаксис уже создает предпосылки решения проблемы расширяемости за счет возможности применения предикатов из библиотечных модулей, имена которых определяются элементом **имя_модуля**. В конкретной реализации, например, для операционной системы Windows, в качестве библиотечного модуля может использоваться стандартная динамически подгружаемая библиотека (DLL), содержащая экспортируемые функции, реализующие предикаты. Соответственно, такие функции возвращают логические значения, отражающие успешность унификации, а параметрами этих функций являются:

а) число параметров предиката N;

б) массив числовых значений (из N элементов), каждое из которых определяет, передается параметр по ссылке (входо-выходной параметр), или является входным, или представляет собой произвольное значение «_»;

в) массив строковых значений (из N элементов), которые представляют текущее содержимое параметров (при этом числа преобразуются в строки).

Отметим, что характерным примером библиотечного предиката является предикат

Predicates.BAL(F,Terminators),

который принимает строку F и определяет, является ли она сбалансированным по всем видам скобок выражением, завершающимся одним из символов строки Terminators или концом строки F. Такой предикат активно используется в шаблонах индуцирующих классов распознавания программ (в целях реконструирования исходной породившей программу ОСМ или решения иных схожих задач) для выделения фрагментов выражений или законченных выражений.

Замечание о полиморфизме предикатов. С точки зрения конкретного предиката *разделение параметров* на входные и входо-выходные (по ссылке) *является достаточно условным*. Постулируем подход, характерный для языков класса Prolog: *предикаты полиморфны* и конкретная форма их исполнения существенно зависит от того, какие из параметров заявлены как входные, какие – как входо-выходные и какие – как произвольные значения.

Все *встроенные предикаты, не загружаемые из библиотечных модулей*, делятся на пять типов, представленные в таблице 2.1.

Таблица 2.1. Типы встроенных предикатов

Тип предиката	**Режимы работы**				**Назначение**
	Добавление	**Удаление**	**Обучение**	**Проверка**	
fast	+	+	-	+	Работа с простыми таблицами, представленными в CSV-формате, значения в таблице ищутся по строгому соответствию. Каждой строке сопоставлен факт, каждому столбцу соответствует один параметр предиката. Порядок параметров тождественен порядку столбцов.
nearest	+	+	-	+	Работа с простыми таблицами, представленными в CSV-формате, значения в таблице ищутся по нестрогому соответствию (с допуском D): для чисел – по допустимому отклонению ±D, для строк – по максимальному расстоянию Левенштейна D. Каждой строке сопоставлен факт, каждому столбцу соответствует один параметр предиката. Порядок параметров тождественен порядку столбцов.
neuro	-	-	+	+	Работа с глубокой нейронной сетью

					прямого распространения с K входов и с одним выходом[1]. В начале списка параметров указываются K значений, соответствующих входам, последний (K+1)-й элемент списка соответствует выходу.
xpath	-	-	-	+	Имеет формат: **xpath(T-переменная, X, V)** Выполняет указанный XPath-запрос X к содержимому T-переменной, полученному в результате работы трансформирующего слоя [грамматического разбора]. Если параметр V – входной, то предикат истинен в случае, если значение V тождественно результату запроса. Если V – параметр по ссылке (входо-выходная переменная), то в него просто помещается результат запроса, причем предикат считается истинным. Если в качестве V указано «_», то запрос просто исполняется и предикат истинен (такой режим может быть полезен в случае, если используются побочные эффекты XPath-запроса, например его расширенные алгоритмические возможности).
xpathf	-	-	-	+	Имеет формат: **xpathf(T-переменная, Name [, Arg1, …, ArgN], V)** Выполняет указанную пользовательскую XPath-функцию с именем Name на XML-содержимом T-переменной, полученным в результате работы трансформирующего слоя

[1] Это ограничение свойственно лишь текущей конкретной реализации. С теоретической точки зрения ничто не мешает работать с сетями более чем с одним выходом.

					[грамматического разбора]. Здесь Arg1…ArgN – аргументы (входные/входо-выходные параметры или «_»), которые, соответственно, могут как передавать значения в функцию, так и принимать из нее значения. Истинность такого предиката как и предиката **xpath**, определяется в зависимости от параметра V, который интерпретируется одинаково в обоих предикатах (см. выше).

В режиме добавления запись предиката, фактически, означает указание значений, которые должны сформировать новую строку соответствующей CSV-таблицы. Соответственно, параметры здесь только входные (не входо-выходные и не пустые значения «_»). Например, если CSV-таблица состоит из трех столбцов, а связанный с ней предикат **pred** имеет тип **fast** или **nearest**, то запись

(фрагмент_рег_выражения)+=>{pred(P.Lemma,"String",12)}

вставит в таблицу строку, включающую значение переменной Lemma регулярно-логического выражения P, строковое значение «String» и числовое значение 12.

В режиме удаления запись предиката, фактически, означает шаблон, включающий конкретные значения или произвольные значения «_», по соответствию которому находятся и удаляются строки соответствующей CSV-таблицы. Соответственно, параметры здесь не могут быть входо-выходными. Например, если CSV-таблица состоит из трех столбцов, а связанный с ней предикат **pred** имеет тип **nearest**, то запись

(фрагмент_рег_выражения)-=>{pred(_,$,12)}

удалит из таблицы строки, у которых первый столбец имеет любое значение, второй столбец содержит значение, близкое по Левенштейну к текущему значению фрагмента регулярного выражения (в круглых скобках), а третий столбец содержит числовое значение, близкое к 12.

В режиме проверки запись предикатов типов **fast** и **nearest** означает шаблон запроса, включающий любые типы значений. Параметры с произвольными значениями «_» игнорируются. Система находит в соответствующей CSV-таблице первую строку, которая содержит указанные значения входных параметров в столбцах, соответствующих этим параметрам. Если такая строка найдена, то входо-выходные параметры получают значения из столбцов, соответствующих этим параметрам.

Запись предиката типа **neuro** *в режиме проверки* может иметь три формы:

а) если все *параметры являются входными*, то сеть просто проверяет, действительно ли указанное значение последнего (K+1)-го параметра равняется выходу сети при указанных значениях K входов (K начальных параметров);

б) если *последний (K+1)-й параметр является входо-выходным*, то сеть вычисляется по указанных значениям K входов (K начальных параметров) и округленное до целого числа значение выхода помещается в (K+1)-й параметр;

в) если *(K+1)-й параметр является входным* и имеет значение F и некоторое подмножество IN множества начальных K параметров является подмножеством входных параметров, а некоторое подмножество OUT – подмножеством входо-выходных параметров и произвольных значений, то сеть пытается решить систему нелинейных уравнений вида

$$NET(IN,OUT) = F$$

относительно входов, соответствующих OUT, и помещает полученные значения, округлив их до целых чисел, в параметры OUT. Для решения системы нелинейных уравнений используется метод Ньютона [27], имеющий квадратичную сходимость. При этом, *в параллельном режиме* проводится несколько запусков процесса поиска OUT с различными начальными точками, из полученных результатов выбирается имеющий минимальную среднеквадратичную ошибку.

Например, если CSV-таблица состоит из трех столбцов, связанный с ней предикат **pred** имеет тип **fast**, а предикат **net** имеет тип **neuro** (например, сеть с тремя входами и одним выходом), то записи

()->{A}

()->{B}

(фрагмент_рег_выражения)?=>{pred(_,"String",$A),net(A,11,5,$B)}

будут интерпретироваться следующим образом:

1. В текущем регулярно-логическом выражении вводятся переменные A и B.

2. В CSV-таблице, связанной с **pred**. осуществляется последовательный поиск строки, у которых первый столбец имеет любое значение, а второй столбец содержит значение, равное строке «String». Если такой строки нет, то унификация указанного микровыражения из двух предикатов останавливается и признается неуспешной. Если такая строка есть (поиск завершается на первой же найденной строке), то значение из ее третьего столбца помещается в переменную A текущего регулярно-логического выражения, после чего переходим к шагу 3.

3. Определяются значения входов нейронной сети: текущее значение переменной A, число 11, число 5. Вычисляется отклик сети, полученное значение помещается в переменную B. Эта операция всегда успешна, поэтому унификация всего микровыражения признается успешной.

Режим обучения имеет смысл только для предикатов типа **neuro** и, фактически, используется как для *создания/инициализации сети* (для запуска такого режима предикат получает *специальные параметры*), так и для выполнения *серии эпох обучения* (для запуска этого режима предикат указывается *без параметров*). Необходимо сказать несколько слов о параметрах предиката при создании сети – они указываются в следующем порядке:

1. Строка в кавычках, представляющая собой *ссылку на предикат* типа **fast** или **nearest**, из CSV-таблицы которого будут браться данные для обучения сети. Каждой строке таблицы соответству-

ет одна обучающая пара. *Столбцы* соответствуют значениям входов и выходов, *нумеруются с нуля*.

2. Строка в кавычках, имеющая вид:

«[» номер_столбца_таблицы {«,» номер_столбца_таблицы} «]»

в которой указаны номера столбцов, из которых последовательно будут браться *значения входов*.

3. Строка в кавычках, имеющая вид:

«[» номер_столбца_таблицы {«,» номер_столбца_таблицы} «]»

в которой указаны номера столбцов, из которых последовательно будут браться *значения выходов*. Напомним, что в текущей реализации поддерживаются только одновыходные сети, но теоретических запретов на наличие множества выходов нет.

4. Строка в кавычках, имеющая вид:

«[» число_нейронов «,» число_нейронов {«,» число_нейронов} «]»

в которой указывается конфигурация сети прямого распространения: из нее последовательно будут браться *количества нейронов в слоях, начиная с входного и заканчивая выходным*.

В текущей реализации поддерживаются *сети с не менее чем двумя слоями*. Выходной слой всегда является линейным, прочие слои – всегда имеют нейроны с активационной функцией «экспоненциальная сигмоида» [16].

Обучение сети проводится стандартным методом обратного распространения ошибки.

Например, если существует **fast**-предикат **pred**, связанный с CSV-таблицей из пяти столбцов (первые четыре – входы сети, пятый – выход), а также предикат **net** типа **neuro** то запись

(фрагм_рег_выражения)*=>{net("pred","[0,1,2,3]","[4]","[5,3,1]"), net)}

будет интерпретироваться следующим образом:

1. Создать нейронную сеть на базе CSV-таблицы предиката **pred**, связать четыре входа сети, соответственно, с нулевым, первым, вторым и третьим столбцами, а выход – с четвертым столбцом. Сеть будет иметь 5 нейронов во входном слое, 3 нейрона в промежуточном и 1 в выходном слое.

2. Провести серию эпох обучения построенной сети.

2.1.4. XPath-подобный алгоритмический язык

Как уже упоминалось, выходная информация трансформирующего слоя (в частности, предоставляемого библиотекой Link Grammar) существует в виде виртуального XML-документа (для обеспечения единообразного формата данных в системе). В соответствии с общей концепцией данной работы выше было принято решение использовать в качестве средств извлечения данных из такого слоя XPath-подобный запросо-алгоритмический язык, дополнив стандартный XPath несколькими конструкциями. Вводимые в XPath алгоритмические возможности существенно расширят функциональность XPath-подобных слабых ограничений (правил) системы прямого логического вывода, которая будет рассмотрена далее.

В-целом, стандартный XPath уже имеет доступные для чтения переменные (записываемые в формате $ИМЯ) и обладает элементарными возможностями последовательного и ветвящегося исполнения серий логических функций (при условии, что используется стандартная концепция сокращенного исполнения логических выражений). В самом деле, XPath-выражение вида

A(…) and B(…) and C(…)

может трактоваться как линейный фрагмент программы, подразумевающий последовательный вызов функций A(…), B(…) и C(…), при условии, что данные функции возвращают истинное логическое значение. Аналогично, выражение

D(…) and E(…) or F(…)

может рассматриваться как условный оператор, который исполняет либо функцию E(…), если результат D(…) истинен, либо функцию F(…), если результат D(…) ложен.

Таким образом, чтобы покрыть базовые потребности программирования, необходимо ввести возможность программного определе-

ния новых функций XPath (если при этом разрешить рекурсию, то с помощью функций можно реализовать еще и циклы) и ввести, как минимум, одну новую стандартную функцию (результат исполнения которой всегда истинен)

«**set**» «(» «**$**» имя_переменной «**,**» выражение «)»

которая позволит присваивать значение выражения переменной.

Предлагается следующий синтаксис программно определяемой функции:

функция = «*****» имя_функции «(» [список_параметров] «)» XPath-выражение «**.**»

имя_функции = идентификатор

список_параметров = параметр {«**,**» параметр}

параметр = параметр_по_ссылке | параметр_по_значению

параметр_по_ссылке = «**&**» «**$**» идентификатор

параметр_по_значению = параметр_указатель | параметр_не_указатель

параметр_указатель = «**$**» «**#**» {«**#**»} [идентификатор]

параметр_не_указатель = «**$**» идентификатор

где XPath-выражение является телом функции, причем его значение определяет результат исполнения функции (возвращаемое значение также может быть определено в теле функции путем присваивания значения встроенной псевдопеременной $result). Параметры могут передаваться как по ссылке, так и по значению, их имена всегда префиксуются символом «**$**». Имена числовых, строковых и логических параметров могут быть произвольными идентификаторами, имена параметров-указателей на отдельные фрагменты XML-документа дополнительно префиксуются одним или несколькими символами «**#**» (при этом для получения элемента по такому указателю в теле функции используется его имя с префиксами «**#**», но без начального знака «**$**»). Вызов программно определяемой функции оформляется таким же образом, как и вызов любой стандартной XPath-функции. Такие функции могут вызываться из регулярно-логических выражений (прямо, в

микропредикате xpathf, или косвенно, в запросе микропредиката xpath).

Приведем два примера программно определяемой функции.

Рекурсивная функция loop($I, $max, $body) исполняет в цикле XPath-выражение, записанное в строковом параметре $body, цикл выполняется по переменной-счетчику $I, от ее переданного начального значения до значения $max включительно.

```
* loop($I,$max,$body) ($I <= $max and set($I, $I+1) and ev-
      al($body) and loop($I,$max,$body)) or true().
```

Здесь фигурирует еще одна новая стандартная функция:

«**eval**» «(» строковое_выражение «)»

которая исполняет переданное в нее строковое выражение, интерпретируя его как XPath-выражение. Вычисленное значение этого выражения является результатом функции eval.

Рекурсивная функция depth(&$OUT1) помещает в свой параметр $OUT1 значение максимальной глубины поддерева XML-тэгов, начиная с точки документа, в которой была вызвана данная функция:

```
 * depth(&$OUT1) set($OUT1,0) and (self::*[set($OUT1,1) and
*[depth($OUT0) and set($OUT1,max($OUT0+1,$OUT1))]] or true()).
```


Следует заметить, что, для полноценности, такому микроязыку программирования необходима также поддержка динамических структур данных – как минимум их создания и удаления. Такая возможность не имеет особого смысла при попытке применения в регулярно-логических выражениях для содержимого T-переменных, поскольку оно передается в вызывающие микропредикаты xpath/xpathf без права изменения, но весьма вероятно применение таких возможностей в XPath-подобных слабых ограничениях (конструирующих правилах) машины прямого логического вывода.

Соответственно, целесообразен ввод еще трех новых функций:

«**create**» «(» XPath-запрос «)»

которая создает новые элементы XML-документа путем применения XPath-запроса в конструирующем смысле (такой режим будет рассмотрен далее, также см. работу [11]),

«**delete**» «(» XPath-запрос «)»

которая удаляет элементы XML-документа, которые возвращает XPath-запрос,

«**transaction**» «(» выражение «)»

которая выполняет выражение (содержащее create/delete-утверждения) в едином, атомарно принимаемом или отменяемом блоке. Если выражение истинно, то транзакция принимается, иначе – отменяется.

В качестве примера приведем набор функций, которые позволяют создать в XML-документе список путем конкатенации двух уже существующих списков (список представляет собой цепочку вложенных тэгов list с атрибутами data [в которых хранятся значения соответствующих элементов списка]):

```
          *concat_list($#,$##) add_list(#/self::*) and
                      add_list(##/self::*).
    *add_list($#) count(list) = 0 and copy_list(#/self::*) or
              list[add_list(#/self::*)] or true().
     *copy_list($#) count(#/list) = 0 or create(list[@data =
     #/list/@data]) and (list[copy_list(#/list)] or true()).
```

Вызов функции конкатенации для документа

```
<a>
      <b>
            <list data="1">
                  <list data="2">
                  </list>
            </list>
      </b>
      <c>
            <list data="3">
                  <list data="4">
                  </list>
            </list>
      </c>
</a>
```

может иметь вид:

```
transaction(concat_list(/a/b,/a/c))
```

при этом вложенные цепочки тэгов list (с атрибутами data) из тэгов, определяемых результатами запросов /a/b и a/c, будут объединены в

общую вложенную цепочку и помещены в точку документа, из которой осуществлялся вызов конкатенации.

И, наконец, дадим реальный пример функции, которая используется в регулярно-логических выражениях в системе PGEN++ в качестве одного из элементов интерфейса со слоем грамматического разбора:

```
 * action(&$VERB,&$OBJ): /*/Link[Name/text()="MVv" and
set($VERB,Left/Value/text()) and set($OBJ,Right/Value/text()) or
     Name/text()="MVIv" and set($OBJ,Left/Value/text()) and
                set($VERB,Right/Value/text())].
```

Данная функция применяется к результатам грамматического разбора некоторого фрагмента текста, выделенного регулярно-логическим выражением, находит в нем глагол, применяемый по отношению к некоторому объекту, и помещает глагол в параметр $VERB, а объект – в $OBJ, в этом случае функция возвращает истину. Если же предложение не содержит указанной грамматической конструкции, функция возвращает ложь. В настоящее время разработано несколько подобных функций, облегчающих взаимодействие микропредикатов регулярно-логических выражений со слоем грамматического разбора в целях выявления смысла исходной естественно-языковой постановки задачи.

Тезис об алгоритмической полноте разработанного микроязыка по Тьюрингу. Разработанный XPath-подобный язык алгоритмически полон по Тьюрингу.

Неформальное доказательство. Сравним описательно-алгоритмические возможности разработанного языка и языка GNU Prolog. Оба языка схожим образом позволяют описать линейные цепочки операторов (вызовов функций или предикатов), ветвящиеся цепочки операторов, циклы (с помощью рекурсии), функции (предикатам Prolog можно поставить в соответствие функции разработанного языка). В обоих языках возможно применение переменных, работа с динамическими данными сложной структуры (базы предикатов и сложные термы в Prolog с одной стороны, и тэги/блоки тэгов в разработанном языке с другой стороны). Соответствие установлено. По-

скольку GNU Prolog является алгоритмически полным по Тьюрингу (на нем можно написать машину Тьюринга, реализующую любой разрешимый по Тьюрингу алгоритм), разработанный язык также является алгоритмически полным по Тьюрингу.

Следствие. Разработанный XPath-подобный язык способен реализовать произвольный разрешимый по Тьюрингу алгоритм обработки данных грамматического разбора.

2.1.5. Конструирующие XPath-запросы

Выше уже упоминалась целесообразность применения синтаксиса XPath в конструирующем смысле. Такая возможность не только представляет существенный теоретический интерес, но и может оказаться достаточно лаконичной и удобной для практического применения.

Лемма о конструирующих XPath-запросах ЛКЗ1. Синтаксис XPath может быть использован для достраивания XML-документа.

Конструктивное доказательство. Введем понятие *конструирующего XPath-запроса* P. Такой запрос синтаксически эквивалентен обычному XPath-запросу Z, с той разницей, что применение P к документу K_1 производит в нем минимально необходимый набор операций достраивания до K_2, таких, чтобы исполнение обычного эквивалентного запроса Z к полученному документу K_2 (по всем дающим положительный ответ вариантам унификации предикатов запроса) было успешным.

Конструирующий запрос P интерпретируется последовательно, слева направо. Если его очередным элементом является имя тэга без предиката, то запрос проверяет наличие соответствующего тэга в указанных местах документа и, если его там нет, создает (с пустыми набором атрибутов и содержанием). Если же очередной элемент – имя тэга Q с предикатом T, то предикат T используется для генерации множества возможных наборов $\{S_i\}$ атрибутов тэга, после чего в ука-

занных местах документа создаются тэги Q, каждый из которых отвечает следующим требованиям: а) набор атрибутов $W \in (\{S_i\} \setminus M)$, где M – множество наборов атрибутов, соответствующих уже присутствующим (в указанных местах документа) тэгах Q; б) на наборе атрибутов W предикат T успешен при его интерпретации по обычным правилам XPath.

Постулируем, что *производящими операциями* в предикатах конструирующих XPath-запросов могут быть только «=», «AND» и «OR». Результатом каждой такой операции является множество значений атрибутов в формате {'A_i=V_i'}.

Результатом операции «@ID=const» является {'@ID=const'}.

Результатом операции «A OR B» является A∪B.

Результатом операции «A AND B» является

$$\begin{cases} A \times B, & \text{если } A \neq \varnothing, B \neq \varnothing; \\ A, & \text{если } B = \varnothing; \\ B, & \text{если } A = \varnothing. \end{cases}$$

Результатом всех прочих операций является ∅.

В качестве примера рассмотрим применение конструирующего XPath в качестве возможного элемента слабых ограничений, выполняющих достраивание первичной ОСМ-модели, представленной в XML-формате, до полноценной смысловой модели. Такой XML-документ имеет фиксированный DTD и описывает *сетевой граф ОСМ – постановки задачи*, вершинами которого являются факты-объекты, каждому из которых соответствует тэг (имя которого совпадает с классом объекта, а атрибуты представляют поля объекта), дочерний по отношению к корню документа OBJS, а связи отражают отношения между объектами и описываются тэгами <I>\<Link> (конец связи) и <O>\<Link> (начало связи), вложенными в тэги фактов-объектов.

Например, если имеем конструирующий запрос вида

```
/OBJS/clsSimpleBlock[(@ID=«HC1» OR @ID=«HC2») AND (@IVar=«C»)],
```

то легко убедиться, что применение вышеприведенных правил дает результирующий документ – ОСМ, который будет обязательно со-

держать следующий фрагмент, описывающий два объекта класса clsSimpleBlock:

```
<OBJS>
      …
      <clsSimpleBlock ID=«HC1» IVar=«C»></clsSimpleBlock>
      <clsSimpleBlock ID=«HC2» IVar=«C»></clsSimpleBlock>
      …
</OBJS>
```

Доказано построением.

2.1.6. Концепция шаблона как переборной группы

Изначально ОСМ в системе PGEN++ обладали способностью *к генерации чередующегося кода*, в котором различные фрагменты относящегося к одному объекту кода могли перемежаться. Такая схема показала достаточную адекватность на ряде задач [6, 9, 11, 14]. Следуя элементарной логике, можно заключить, что восстановление ОСМ того или иного вида по произвольному тексту также должно *принимать во внимание возможное чередование* его фрагментов, относящихся к одному и тому же объекту. Приходим к выводу, что шаблон Т некоего индуцирующего метода должен обладать *способностью к унификации целой группы регулярно-логических выражений*, которые, в общем случае, могут либо примыкать друг к другу, либо находиться на заранее неизвестном расстоянии друг от друга. *Каждый обнаруженный вариант унификации порождает отдельный объект* выходной ОСМ.

Логично предположить, что некоторые фрагменты текста могут однозначно относиться лишь к одному индуцируемому объекту определенного класса (то есть быть *глобально уникальными, захватываемыми*), в то время как некоторые иные фрагменты являются лишь обязательным *контекстом*, который может относиться к множеству объектов различных классов. Соответственно, регулярно-логические выражения (элементы группы, формирующей шаблон), выделяющие фрагменты первого типа, можно назвать *глобально уникальными*, а выделяющие фрагменты второго типа – *контекстными*. Также можно

предположить существование варианта, когда фрагмент является не глобально, а *локально уникальным*, то есть может относиться к объектам разных классов, но в каждом классе – только к одному объекту.

Дополнительно отметим, что, по меньшей мере, для контекстов, имеет смысл, по аналогии с переменными и константами в языках программирования, выделить *модифицируемые* (в логических скриптах M_L) и *немодифицируемые* (там же) регулярно-логические выражения. Для уникальных выражений можно предположить, что они всегда потенциально модифицируемы.

Сведем вышесказанное в таблицу 2.2.

Таблица 2.2. Виды выражений группы и их характеристики

Тип выражения	**Название**	**Возможность модификации**	**Возможная унификация**	
			Для классов	**Для объектов**
context	Немодифицируемый контекст	-	Многократная	Многократная
unique	Локально уникальный (с захватом единственного фрагмента для каждого класса)	+	Многократная	Однократная
global_unique	Глобально уникальный (с глобальным захватом фрагмента)	+	Однократная	Однократная
nonunique	Модифицируемый контекст	+	Многократная	Многократная

Пояснения. Каждый обнаруженный вариант унификации (объект выходной ОСМ), содержащий хотя бы одно глобально уникальное выражение получает *глобальный уникальный идентификатор*, который генерируется заранее и всегда доступен в регулярно-логическом выражении в таблице **db_gid.csv** (хранящей единственное актуальное числовое значение уникального идентификатора), например, через

предикат типа **fast**. Список значений, равных уникальным идентификаторам унифицированных фрагментов (в порядке их следования в исходном тексте), доступен в логическом скрипте M_L шаблона через предикат-факт **global_trace(TR)**, который унифицирует параметр TR с этим списком.

В логическом скрипте M_L, который вызывается для каждого варианта успешной унификации, *глобальный уникальный идентификатор текущего варианта доступен* через предикат-факт **global_id(GID)**, который унифицирует GID с этим идентификатором. При этом результаты разбора регулярно-логическими выражениями отображаются в дерево переменных/значений (представляется в формате XML), которое доступно в скрипте через XPath-интерфейс.

Несколько сложнее организовано *взаимодействие со скриптами* M_{LD}, поскольку они, для упрощения архитектуры системы и самих скриптов, не имеют доступа к полученному дереву переменных/значений регулярно-логических выражений, а работают с некоторой первичной входной ОСМ в XML-формате. Соответственно, такая ОСМ формируется последовательно, для каждого варианта унификации шаблона (нового объекта-тэга) срабатывает специальный оператор **@auto**, для которого указывается серия XPath-запросов к дереву переменных/значений и имена полей нового объекта (атрибутов тэга), которые заполняются результатами выполнения этих запросов, и/или константами, и/или результатами специальных встроенных функций. Допускается и «перекачка» результата запроса в виде XML-фрагмента непосредственно в тело тэга нового объекта, для этого в качестве заполняемого поля указывается специальный идентификатор «**InnerDoc**».

Особо рассмотрим вариант, когда индуцирующий метод некоего класса с идентификатором A имеет *непустой шаблон T и пустой логический скрипт* $M_L = \varepsilon$. В таком случае, при унификации шаблона T, в список TR попадает не собственно значение уникального идентификатора ID, а сложный терм вида gid(A,ID). Это сделано для того, чтобы дать логическим скриптам прочих классов информацию о том, к

какому классу имеет отношение такой унифицированный, но необработанный фрагмент.

Дополнительно заметим, что если шаблон не содержит глобально уникальных регулярно-логических выражений, то соответствующие ему варианты унификации получают неуникальный фиктивный идентификатор -1.

Учитывая, что выражения одной группы-шаблона могут, в конечном итоге, порождать множество объектов (по объекту для каждого успешного варианта унификации), приходим к выводу о том, что *группа, в общем случае, должна быть полно-переборной* (комбинаторно перебираются все варианты успешных унификаций ее элементов), причем для каждого ее элемента-выражения следует указывать *допустимое количество унификаций* относительно каждой унификации предыдущего элемента, расположенного в группе левее (на практике такое количество обычно принимает либо значение «единичное срабатывание», либо «произвольное число срабатываний»). Поясним эту идею на *примере*. Пусть имеется группа (A,B,C), причем допустимыми количествами унификации являются (2,3,1). Тогда имеем следующие *варианты перебора*:

(A1,B1,C1), (A1,B2,C2), (A1,B3,C3), (A2,B4,C4), (A2,B5,C5), (A2,B6,C6).

Далее заметим, что на практике все варианты перебора используются далеко не всегда, поскольку действует следующее *правило*:

если очередная унификация некоего элемента прошла неуспешно, то все варианты, являющиеся производными по отношению к текущему, *отсекаются*.

Для предыдущего примера это означает, что если унификация A2 неуспешна, то отсекаются все сопутствующие варианты, и в результате остаются лишь:

(A1,B1,C1), (A1,B2,C2), (A1,B3,C3).

Итак, *шаблоны – переборные группы модифицированных регулярных выражений*. Шаблоны, фактически, являются распознающе-переборным механизмом, который последовательно применяется ко

входному тексту с определением его фрагментов, соответствующих шаблону (каждому такому фрагменту будет соответствовать распознанный первичный объект выходной модели).

Определенные сложности возникают при *неполной унификации*, которая признается *неуспешной*. Например, в группе (A,B,C) могло успешно унифицироваться выражение A, но неуспешно B, в результате унификация C не проводится, а унификация всей группы неуспешна. Проблема состоит в том, что выражение A могло модифицировать (с помощью предикатов типа **fast/nearest**) часть CSV-таблиц и эти изменения при неуспешной унификации следует откатить. Решение заключается в том, что, фактически, *каждая попытка унификации группы неявно заключается в транзакцию*, которая принимается только при полной успешной унификации рассматриваемого варианта. С этой целью предикаты **fast/nearest** реализуются с *журналированием всех добавлений и удалений*.

2.1.6.1. Синтаксис распознающей части индуцирующих скриптов

Подводя некоторые итоги сказанному выше о шаблонах T, предложим следующий *простой синтаксис распознающей части индуцирующих скриптов*:

распознающая_часть = {оператор_шаблона}
оператор_шаблона = указание_версии | декларация_предиката | элемент_группы | склейка_элементов_группы | замена_элемента | автозаполнение_полей_объекта | комментарий
ПС = переход_на_следующую_строку
указание_версии = «**@versions(**» имя_версии «**,**» имя_версии «**)**» ПС
декларация_предиката = предикат_таблицы | предикат_нечеткой_таблицы | предикат_нейросети
предикат_таблицы = «**@fast(**» идентификатор «**,"**» имя_CSV_файла«"**).**» ПС
предикат_нечеткой_таблицы = «**@nearest(**» идентификатор «**,**» допустимая_погрешность_сравнения «**,"**» имя_CSV_файла«"**).**» ПС

предикат_нейросети = «**@neuro(**» идентификатор «**,"**»
имя_файла_с_коэффициентами_сети «**").**» ПС
элемент_группы = модиф_контекст | немодиф_контекст |
глоб_уник_выражение | лок_уник_выражение
модиф_контекст = «**nonunique(**» параметры_рег_выражения «**):-**»
регулярно_логическое_выражение «.» ПС
немодиф_контекст = «**context(**» параметры_рег_выражения «**):-**»
регулярно_логическое_выражение «.» ПС
глоб_уник_выражение = «**global_unique(**» параметры_рег_выражения
«**):-**» регулярно_логическое_выражение «.» ПС
лок_уник_выражение = «**unique(**» параметры_рег_выражения «**):-**»
регулярно_логическое_выражение «.» ПС
параметры_рег_выражения = идентификатор_выражения «**,**»
допустимое_число_унификация
допустимое_число_унификаций = число | «**infinity**»
склейка_элементов_группы = «**glue:-.**» ПС
замена_элемента = «**replace(**» идентификатор_выражения «**):-**»
заменяющее_выражение «**.**» ПС
автозаполнение_полей_объекта = «**@auto:-**» автозаполнение {«**,**»
автозаполнение} «**.**» ПС
автозаполнение = авто_элемент | необязательный_авто_элемент
авто_элемент = авто_выражение {«**+**» авто_выражение} «**=>**»
имя_поля
необязательный_авто_элемент = «**[**» авто_элемент «**]**»
авто_выражение = константа | функция | запрос
имя_поля = «**"**» идентификатор «**"**»
константа = «**"**» произвольный_текст «**"**»
функция = «**random()**» | «**randomid()**»
запрос = идентификатор_выражения «**:**» «**"**»
XPath_запрос_к_результату «**"**»
комментарий = «**%**» [строка] ПС

В целом, данное описание синтаксиса вполне отвечает принципу самодокументируемости, дадим лишь некоторые пояснения:

1. Замещающее_выражение в замене элемента представляет собой простую строку текста, которая может содержать несколько макросов:

${имя_переменной} – заменяется на текущее значение переменной выражения, идентификатор которого указан в макросе;

$(имя_файла) – заменяется на содержимое файла с указанным именем.

2. Директива указания принадлежности версии **@versions** позволяет разбить шаблон на несколько частей, каждая из которых включается в одну или несколько именованных версий. Директива действует на те части шаблона, которые начинаются сразу после нее и заканчиваются со следующей директивой или с концом индуцирующего скрипта. В системе PGEN++ при выборе множества индуцирующих классов собирается (из индуцирующих скриптов) информация о версиях, после чего пользователю предлагается выбрать из них рабочую.

3. Функция **random()** генерирует случайное число, функция **randomid()** генерирует случайный идентификатор.

Далее приведен пример распознающей части скрипта класса clsSimpleInput ввода данных с клавиатуры (из набора классов, выполняющих распознавание программы или постановки задачи обработки векторных данных). Скрипт содержит версии Programmatical (задача сформулирована в виде порожденной программы), Russian (задача сформулирована в виде постановки на русском языке, вариант без применения трансформирующего слоя) и RussianGrammar (задача сформулирована в виде постановки на русском языке, используется трансформирующий слой грамматического разбора):

```
@versions(Programmatical)
 @global_unique(PROCESS,infinity):-
  (\s*for\s*\(i\s*\=\s*0\;\s*i\s*\<\s*(\d+)->
{N}\;\s*i\+\+\)\s*\{\\n\s*printf\(\"(\w+)->
{VECTOR}\[\%i\]\s*\=\s*\"\,\s*i\)\;\\n\s*scanf\(\"\%lf\"\,\s*\&(\w+)==
>{VECTOR}\[i\]\)\;\\n\s*\}\\n)|(\s*printf\(\"(\w+)->
{SCALAR}\s*\=\s*\"\)\;\\n\s*scanf\(\"\%lf\"\,\s*\&(\w+)==>{SCALAR}\)\;
\\n).
```

```
@versions(Russian,RussianGrammar)
 @nearest(db_vvedem,2,"db_vvedem.csv").
 @fast(db_var,"db_var.csv").
@versions(Russian)
 @global_unique(PROCESS,infinity):-
  (()->{VAR}([А-Яа-я]+)?=>{db_vvedem($)}\s+((с\s+клавиатуры\s+)?)-
>{KEYB}
   ([А-Яа-я]+)?=>{db_var($,$VAR)}\s+(\w+)->{ID}
   ((\s+с\s+клавиатуры)?)!=>{KEYB}\s*\.).
@versions(RussianGrammar)
 @nearest(db_klav,2,"db_klav.csv").
@context(PREV,infinity):-
  (((^)|(\.)+)(\s*\\n)*)\s*.
@glue:-.
@global_unique(PROCESS,1):-
  ()->{VERB}()->{OBJ}()->{ATTR}()->{VAR}(([^.A-Za-z]+(\w+)->{ID}[^A-
Za-z.]*\.)->
{Grammar.ru{SENTENCE}}(*PRUNE))?=>{
xpathf(SENTENCE,'action',$VERB,$OBJ,_),db_vvedem(VERB),db_var(OBJ,$VAR
),
xpathf(SENTENCE,'obj_attr',OBJ,$ATTR,_),db_klav(ATTR)
}.
@versions(Russian,RussianGrammar)
 @auto:- PROCESS:"//ID/text()" => "IVar", "in"+PROCESS:"//ID/text()"
=> "ID".
```


Здесь используется несколько предикатов, работающих с таблицами:

db_vvedem – с нечеткой таблицей синонимов к слову «введем»,

db_var – с таблицей названий видов переменной (вектор или скаляр),

db_klav – с нечеткой таблицей синонимов к слову «клавиатура».

В версии RussianGrammar использованы предикаты вызова алгоритмических XPath-функций action (находит в разобранном предложении грамматическую связь действия VERB над объектом OBJ) и obj_attr (находит в разобранном предложении грамматическую связь объекта OBJ с атрибутом действия ATTR).

Далее приведем пример выделенной первичной ОСМ, которая построена распознающими классами для задачи простой обработки векторных данных, поставленной на естественном языке:

Ввести скаляр max. Введем вектор V из 10 элементов. Зададим вектор V с клавиатуры. Найдем также максимум вектора V и поместим результат в скаляр max.
Вывести скаляр max на экран. А среднее арифметическое считать не будем. Тест: "1 2 3 4 5 6 7 8 9 10" дает "V[0] = V[1] = V[2] = V[3] = V[4] = V[5] = V[6] = V[7] = V[8] = V[9] = max = 10.000000".

Соответствующая ОСМ, представленная в XML-формате, выглядит следующим образом:

```
<OBJS>
      <clsSimpleScalar WORDF="0" WORDN="2" GID="0" ID="max" >
            <I ID="Asgn" Ref="199523925"></I>
            <O ID="Handle" Ref="74237061"></O>
      </clsSimpleScalar>
      <clsSimpleVector WORDF="2" WORDN="8" GID="1" ID="V" Size="10" >
            <I ID="Asgn" Ref="173052978"></I>
            <O ID="Handle" Ref="45800782"></O>
      </clsSimpleVector>
      <clsSimpleMat  WORDF="8"  WORDN="12"  GID="4"  Op="Max"  IVar="V"
OVar="max" ID="MaxVmax" >
            <I ID="Arg" Ref="59606934"></I>
            <I ID="Prev" Ref="33135987"></I>
            <O ID="Next" Ref="32025147"></O>
            <O ID="Res" Ref="67486572"></O>
      </clsSimpleMat>
      <clsSimpleInput  WORDF="12"  WORDN="18"  GID="5"  IVar="V"  ID="inV"
>
            <I ID="Arg" Ref="150738525"></I>
            <I ID="Prev" Ref="44702149"></I>
            <O ID="Next" Ref="73095703"></O>
      </clsSimpleInput>
      <clsSimpleOut    WORDF="18"    WORDN="24"    GID="6"    IVar="max"
ID="outmax" >
            <I ID="Arg" Ref="87432861"></I>
            <I ID="Prev" Ref="58612061"></I>
            <O ID="Next" Ref="68634033"></O>
      </clsSimpleOut>
      <clsSimpleTerminator WORDF="24" WORDN="24" GID="7" ID="END" >
            <I ID="End" Ref="108935547"></I>
      </clsSimpleTerminator>
      <clsSimpleTest WORDF="24" WORDN="24" GID="8" Input="1 2 3 4 5 6
7 8 9 10" Output="V[0] = V[1] = V[2] = V[3] = V[4] = V[5] = V[6] =
V[7] = V[8] = V[9] = max = 10.000000" ID="nsprtdy" >
```

```
    </clsSimpleTest>
</OBJS>
```

Выводы ко второй главе

В данной главе предложены распознающие объектно-событийные модели.

Предложены два способа интерпретации распознающих ОСМ для структурированных текстов программ и неструктурированных/слабо структурированных текстов (например, на естественном языке) с восстановлением линейных ОСМ, являющихся первичными описаниями структуры/смысла текста. При этом используется новая концепция шаблонов как переборных групп из контекстных и основных модифицированных регулярно-логических выражений, имеющих простой и доступный интерфейс (в виде специальных предикатов) с элементарными CSV-базами данных (БД), с трансформирующим слоем грамматического разбора и, в некоторых случаях, с глубокими нейронными сетями прямого распространения. Такой подход позволяет существенно упростить восстановление первичных линейных ОСМ за счет потенциального переноса части нагрузки из программы в данные (БД, трансформирующий слой и нейросети).

На базе синтаксиса XPath предложен простой алгоритмический микроязык, для которого показана алгоритмическая полнота. Введено понятие конструирующего XPath-запроса, являющегося одним из средств данного микроязыка, определены синтаксис и семантика предложенных алгоритмических конструкций. Предложена схема применения микроязыка в регулярно-логических выражениях при оперативной логико-алгоритмической обработке первичных результатов разбора входного текста (полученных как с привлечением, так и без привлечения грамматического разбора).

Глава 3. Трансформация первичных ОСМ. Обратный и прямой логический вывод

Целью данной главы является рассмотрение вопросов восполнения построенных первичных ОСМ до полноценных смысловых моделей, а также вопросов автоматизированного приобретения знаний, необходимых для такого восполнения.

Для реализации данной цели поставим следующие задачи:

1. Сформулировать задачу восполнения (достраивания) ОСМ.

2. Предложить синтаксис и семантику средств обратного и прямого логического вывода-восполнения.

3. Предложить адекватную поставленным целям машину прямого логического вывода и все необходимые для ее функционирования формализмы.

4. Предложить механизмы автоматизированного приобретения знаний, необходимых для распознавания и восполнения ОСМ, в заданной предметной области.

5. Провести апробацию описанных в данной работе методов и средств для трех основных задач: верификации порождающих программу скриптов (на базе распознавания модели программы), генерации решающих программ по естественно-языковым постановкам и интеллектуальной трансформации программ.

3.1. Общее понятие о достраивании ОСМ

Как уже упоминалось выше, результатом унификации шаблонов Т индуцирующих методов является первичная ОСМ, представляющая собой базовую совокупность фактов из постановки задачи. Такая ОСМ может быть представлена как набором фактов динамической базы данных GNU Prolog, так и XML-документом. Дальнейшая интеллектуальная обработка первичной модели состоит в применении

логических скриптов, выполняющих трансформирующее достраивание первичной ОСМ до полной выходной ОСМ, в частности:

а) добавление отсутствующих фактов-объектов, наличие которых подразумевается постановкой задачи;

б) добавление фактов-объектов, представляющих расширенный план решения поставленной задачи;

в) установка всех необходимых связей между объектами.

Данная задача может быть решена как обратным, так и прямым логическим выводом. При этом обратный вывод дает результат быстрее, но требует не только специальных знаний по логическому программированию, но и точного представления о цели достраивания-доказательства. Прямой вывод более трудоемок, но, как будет показано далее, позволяет не задавать цель в явном виде (при необходимости, если возможны несколько успешных вариантов достраивания, дополнительно определяются тесты, которые должна проходить программа, построенная по ОСМ-результату достраивания) и описывать задачу набором абсолютно декларативных локальных правил.

3.2. Достраивание модели обратным логическим выводом

В данном случае используется представление ОСМ в виде набора фактов динамической базы данных GNU Prolog. Соответственно, логические правила трансформации представляются скриптами M_L, представляющими собой набор предикатов, записанных на языке GNU Prolog. Каждый такой скрипт анализирует результаты унификации соответствующего ему шаблона, извлекая значения введенных переменных из группы регулярно-логических выражений, формирующих шаблон, и анализируя содержимое динамической базы фактов, являющейся общей для всех таких скриптов, пополняемой ими и хранящейся на протяжении всего сеанса вывода выходной ОСМ. Поскольку модель, обычно, имеет прямое отображение в набор динами-

ческих фактов, соответственно как ее порождение, так и ее трансформации выполняются путем манипулирования такими фактами в базе. Таким путем может быть получено решение: а) обратной задачи системы PGEN++, заключающейся в восстановлении по конечной порожденной программе модели, ее породившей, или б) задачи построения ОСМ (в XML-представлении), выражающей план решения некоей задачи, из текста на естественном языке [представляющим хотя бы фрагменты такого плана], руководствуясь некоторыми логическими правилами трансформации, заложенными в скрипты M_L программистом системы.

Напомним, что для каждого класса A скрипт M_L вызывается при каждом успешном варианте унификации шаблона T (фактически, каждый такой вариант соответствует объекту выходной ОСМ), имеющем идентификатор, доступный через предикат **global_id**. Список идентификаторов всех унифицированных фрагментов доступен через предикат **global_trace**. Добавление и удаление динамических фактов выполняется стандартными предикатами **asserta**, **assertz**, **retractall** и другими.

Поскольку скрипт должен иметь доступ к распознанным шаблоном синтаксическим элементам, и с учетом того, что именованные переменные шаблона могут быть вложенными и неоднократно унифицироваться, выше уже отмечалось, что конечным представлением разобранных элементов является дерево переменных/значений с абстрактным корнем, на следующем уровне которого присутствуют переменные, не являющиеся вложенными, далее следует уровень их значений, а далее уровни вложенных переменных и их значений следуют, чередуясь.

Это приводит к достаточно очевидной идее применения синтаксиса по типу *XPath* для доступа к разобранным элементам конструкций. *При этом абстрактный корень имеет имя **root**, а каждая следующая пара уровней (переменные и их значения) рассматривается как один виртуальный уровень дерева*, где переменная с именем D

представляет имя узла D, а к ее значениям можно обратиться указанием вида

D[номер_значения],

где значения нумеруются, начиная с единицы.

XPath-доступ реализуется специальным предикатом

xpath(EXPR,XPATH,RES),

где EXPR – строковое значение, представляющее идентификатор регулярно-логического выражения из текущего шаблона, XPATH – строковое значение, представляющее XPath-запрос, возвращающий RES – список неких внутренних идентификаторов узлов (если запрашиваются узлы дерева) или список текстовых значений (если запрашиваются значения узлов дерева). Отметим, что также возможен режим запуска, при котором заданы все три аргумента – в этом случае система проверяет, действительно ли указанный RES является результатом указанного XPATH для заданного EXPR.

Рассмотрим *пример*. Пусть имеется следующий элемент группы, распознающий серию простых присваиваний значений переменным в С-программе:

```
@unique(Assgn,1):-
  (\s*(\w+)->{Name}\s*\=\s*(\w+)->{Val}\s*\;(\\n)*)->{EXPR}+.
```


Тогда запрос вида

xpath(“Assgn”,”//EXPR[2]/Name/text()”,R)

для входного текста

Query = Now;
Truth = 2018;

унифицирует значение R = ‘Truth’.

3.2.1. Общий синтаксис индуцирующих скриптов

Теперь можно выписать *общий синтаксис индуцирующего скрипта*:

скрипт = {оператор_скрипта}
оператор_скрипта = оператор_шаблона | макропредикат

макропредикат = предикат_Prolog | цель | финализация

предикат_Prolog = заголовок_предиката [«**:-**» тело_предиката] «**.**» ПС

цель = «**@goal:-**» тело_предиката «**.**» ПС

финализация = «**@done:-**» тело_предиката «**.**» ПС

В этой схеме заголовок и тело любого предиката оформляются строго по правилам GNU Prolog. Скрипт может содержать произвольное количество предикатов Prolog и по одному макропредикату цели и финализации. *Предикат цели* содержит описание действий, которые должны быть предприняты сразу после очередной успешной унификации группы содержащихся в скрипте регулярно-логических выражений (шаблона). По окончании исполнении предиката цели закрывается файл вывода GNU Prolog (в него обычно помещается специальная служебная информация, рассмотрение которой выходит за рамки данной работы) и вызывается необязательный *предикат финализации*.

Далее приведен пример индуцирующего скрипта, обнаруживающего во входном тексте множество фрагментов, отвечающих за вывод на экран переменных при решении простых учебных задач обработки векторных данных, и реконструирующего соответствующие объекты выходной ОСМ, представленные в виде динамических фактов **simple_act**. Реконструкция здесь осуществляется в двух версиях: из текста реализующей С-программы (режим **Programmatical**) и из постановки соответствующей исходной учебной задачи на русском языке (режим **Russian**, слой грамматического разбора не используется). Используется таблица синонимов слову «Выведем» (**db_vyvedem.csv**) и таблица вариантов написания слов «Скаляр/вектор» (**db_var.csv**). Первая таблица состоит из единственного столбца с синонимами. Втора таблица включает два столбца, в первом содержатся варианты написания, во втором – тип соответствующей переменной (0 – вектор, 1 – скаляр).

Предполагается, что до запуска этого скрипта уже были распознаны объекты классов «Скаляр» и «Вектор», соответствующая им информация была помещена в динамические факты **simple_scalar** и **simple_vector**. На практике это означает, что в распознающей ОСМ

классы «Скаляр» и «Вектор» структурно предшествуют классу «Вывод».

```
@versions(Programmatical)
@global_unique(PROCESS,infinity):-
  (\s*printf\(\"(\w+)-
>{VECTOR}\s*\=\s*\[\"\)\;\\n\s*for\s*\(i\s*\=\s*0\;\s*i\s*\<\s*(
\d+)-
>{N}\;\s*i\+\+\)\\n\s*printf\(\"\%lf\s*\"\,\s*(\w+)==>{VECTOR}\[
i\]\)\;\\n\s*printf\(\"\]\\\\n\"\)\;\\n)|(\s*printf\(\"(\w+)-
>{SCALAR}\s*\=\s*\%lf\\\\n\"\,\s*(\w+)==>{SCALAR}\)\;\\n).
handle:-xpath('PROCESS','//VECTOR/text()',[VText]),
  xpath('PROCESS','//N/text()',[NText]),
  simple_vector(_,VText,NText),
  global_id(GID),assertz(simple_act(GID,out,VText,'')),!.
handle:-xpath('PROCESS','//SCALAR/text()',[SText]),
  simple_scalar(_,SText),
  global_id(GID),assertz(simple_act(GID,out,SText,'')),!.
@versions(Russian)
 @nearest(db_vyvedem,2,"db_vyvedem.csv").
 @fast(db_var,"db_var.csv").
 @global_unique(PROCESS,infinity):-
  (()->{VAR}([А-Яа-
я]+)?=>{db_vyvedem($)}\s+((на\s+экран(е?)?\s+)?)->{KEYB}
   ([А-Яа-я]+)?=>{db_var($,$VAR)}\s+(\w+)-
>{ID}((\s+на\s+экран(е?))?)!=>{KEYB}\s*\.).
@versions(Russian)
 handle:-xpath('PROCESS','//ID/text()',[VText]),
  xpath('PROCESS','//VAR/text()',['0']),
  simple_vector(_,VText,_),
  global_id(GID),assertz(simple_act(GID,out,VText,'')),!.
 handle:-xpath('PROCESS','//ID/text()',[SText]),
  xpath('PROCESS','//VAR/text()',['1']),
  global_id(GID),assertz(simple_act(GID,out,SText,'')),!.
@versions(Programmatical,Russian)
 @goal:-
  handle,!.
 @done:-
  clear_db.
```

3.3. Достраивание модели прямым логическим выводом

В данном случае используется представление ОСМ в виде виртуального XML-документа. Важно отметить, что если при обратном выводе задача обобщения данных, извлеченных регулярно-логическими выражениями, с формированием первичной ОСМ как набора объектов, возложена непосредственно на скрипты M_L, при прямом выводе первичная ОСМ формируется до запуска скриптов логического вывода M_{LD} (процесс формирования, как и при обратном выводе, в значительной степени задействует XPath-запросы к результатам унификации шаблонов, но выполняется средствами автозаполнения, описанными в пункте 2.1.6.1).

Логические правила трансформации (достраивания) представляются скриптами M_{LD}, совокупность которых, собираемая по мере прохождения по сети распознающих классов, образует общий скрипт M_{DIR}, интерпретируемый специальной машиной прямого логического вывода. Процесс такой интерпретации является достаточно тяжеловесным с вычислительной точки зрения, соответственно актуальным является применение параллельных вычислений и различных оптимизаций процесса вывода.

Особенно существенно интерпретация может быть оптимизирована при решении схожих задач. Поэтому в машину прямого логического вывода помимо первичной ОСМ-документа дополнительно поступает информация о структуре и содержании задачи, описываемая разного рода метриками, получаемыми из промежуточных результатов работы трансформирующего слоя:

а) ключевыми словами, упомянутыми в качестве аргументов и полученными в качестве результатов XPath-функций, вызывавшихся в процессе работы из регулярно-логических выражений шаблонов;

б) частотами ключевых слов;

в) наборами грамматических отношений между вышеуказанными ключевыми словами.

Кроме того, в машину вывода поступает и полная текстовая формулировка задачи, представленная в виде цепочки ключевых слов в порядке их упоминания (такие цепочки для разных задач могут сравниваться, например, по расстоянию Левенштейна).

3.3.1. Основной алгоритм машины прямого вывода XML-документа (модели задачи)

С формальной точки зрения возможности достраивания XML-модели полностью описываются грамматикой соответствующего документа, представленной в виде его DTD. Поэтому, требуемые *правила достраивания* будут являться, скорее, *слабыми ограничениями* (поскольку некоторые из них носят рекомендательный характер), в результате применения которых недопустимые варианты документа (выходной модели) бракуются, но можно выделить один или несколько корректных по построению и по семантике вариантов документа. Эти варианты (если их предполагается более одного) проверяются на *наборе заданных тестов*[1] (которые вместе с правилами достраивания составляют *комплект синтаксических средств дополнения и реконструкции XML-документов*, представляющих сетевой граф объектно-событийной модели, отражающей постановку и/или план решения некоторой задачи) и определяются конечные корректные варианты XML-модели. Тесты являются двойками (I, O), где I – строка, представляющая содержимое входного файла программы (построенной системой PGEN++ для решения задачи, описываемой текущим вариантом XML-модели), а O – строка, представляющая содержимое выходного файла. Тест успешен, если программа, построенная системой PGEN++ по текущей XML-модели, успешно компилируется, запуска-

[1] На уровне ОСМ тесты распознаются и описываются как объекты специальных классов.

ется, и, на указанном входном файле выдает выходной файл, содержание которого синтаксически эквивалентно O.

Разделим правила достраивания на *проверочные* (проверяющие некий факт) и *конструирующие* (являющиеся таким расширением проверочных правил, которое способно к частичному достраиванию текущего документа). Каждое правило может быть применено к текущей версии XML-документа *в проверочном режиме*, причем результат применения может быть одним из следующих:

а) *строгое противоречие* – правило не выполняется и никакие достраивания документа не смогут привести к его выполнению;

б) *нестрогое противоречие* – правило не выполняется, но его запуск *в конструирующем режиме* способен породить документ, в котором оно выполнится;

в) *соответствие* – правило выполняется.

Правила имеют *веса*, по умолчанию равные единице. Веса используются при вероятностном определении очередности выбора правила из набора активных правил. Данный вопрос будет рассмотрен далее, пока отметим лишь, что в системе PGEN++ реализована вероятностная модель, использующая информацию об успешных (ранее приведших к корректному результирующему XML-документу) последовательностях применения правил.

Достраивание документа описывается *рекурсивным алгоритмом* ALG(G, E), выполняющим последовательный перебор всех возможных вариантов достраивания E с отсечением тех вариантов, которые некорректны (имеют структуру, невозможную с точки зрения текущего набора классов объектно-событийной модели, соответствующей текущему XML-документу) или входят в строгое противоречие с одним или несколькими правилами. На каждом шаге алгоритма:

1. Имеем текущий вариант документа E, порожденный применением некоторого правила из документа G.

2. Документ E проверяется на корректность по структуре. Если он некорректен, то шаг алгоритма признается неудачным и производится откат к варианту документа G.

3. Для документа E запускаются все имеющиеся правила в проверочном режиме.

3.1. Если все правила сообщили о соответствии, то документ E отправляется в систему PGEN++ на порождение соответствующей программы, решающей поставленную этим документом задачу, после чего полученная программа проходит набор тестов. Если тесты не пройдены, то документ E бракуется и производится откат к варианту G. Если тесты пройдены, то E – результат, алгоритм заканчивает работу.

3.2. Если хотя бы одно правило сообщило о строгом несоответствии, то шаг алгоритма признается неудачным и производится откат к варианту документа G.

3.3. Иначе выделяем подмножество правил, которые сообщили о нестрогом противоречии. Каждое из этих правил запускается в конструирующем режиме, в результате чего получаем вектор новых вариантов документа Y. Последовательность элементов Y_i в векторе определяется с помощью упомянутой выше вероятностной модели. К каждому из вариантов Y_i применяем рекурсивно этот же алгоритм (с откатом к текущему варианту E при неудачах) ALG(E, Y_i).

Остается определить правила – слабые ограничения.

3.3.2. Правила достраивания

Общепризнанным стандартом доступа к XML-документу является язык запросов XPath, поэтому представляется целесообразным взять его за основу как в проверочном, так и в конструирующем случаях. При этом *воспользуемся ранее введенным нами алгоритмическим XPath-микроязыком* для определения и вызова специфических функций из XPath-элементов правил – слабых ограничений. Это потенциально существенно повысит описательно-алгоритмические возможности правил.

Напомним, что XML-документ модели имеет фиксированный DTD и описывает *сетевой граф постановки задачи*, вершинами которого являются факты-объекты, каждому из которых соответствует тэг (имя которого совпадает с классом объекта, а атрибуты представляют поля объекта), дочерний по отношению к корню документа OBJS, а связи отражают отношения между объектами и описываются тэгами <I>\<Link> (конец связи) и <O>\<Link> (начало связи), вложенными в тэги фактов-объектов.

Выделим *5 видов правил*:

1. Конструирующее *правило единственности*. Имеет синтаксис:

[знак] [вес] XPath-выражение «.»

знак = инверсия | «+»

инверсия = «-»

вес = «{» вещественное число «}»

Данное правило в проверочном режиме определяет, возвращает ли указанное XPath-выражение единственный элемент, а в конструирующем режиме достраивает документ, воспринимая XPath-выражение как конструирующее.

2. Проверочное *правило следования*. Имеет синтаксис:

[знак] [вес] «[» XPath-выражение-1 «]» «>>» «[»XPath-выражение-2 «]» «.»

Данное правило вычисляет наборы объектов J_1 и J_2 текущего графа-модели, представленной XML-документом, соответствующих узлам документа, выделяемым запросами XPath-выражение-1 и XPath-выражение-2. Правило возвращает соответствие, если в графе из любой вершины, входящей в J_1, существует путь в какую-либо вершину, входящую в J_2.

3. Проверочное *правило количественного отношения*. Имеет синтаксис:

[знак] [вес] «[» XPath-выражение-1 «]» отношение «[»XPath-выражение-2 «]» «.»

отношение = «<» | «>» | «<=» | «>=» | «=» | «!=»

Данное правило вычисляет XPath-выражение-1 и XPath-выражение-2 с результатами K_1 и K_2 соответственно. Правило возвращает соответствие, если K_1 находится в указанном отношении с K_2, смысл отношения зависит от типов K_1 и K_2.

4. Конструирующее *правило восполнения*.

[знак] [вес] «[» XPath-выражение-1 «]» оператор «[»XPath-выражение-2 «]» «.»

оператор = «=>» | «=>>»

Здесь одно и только одно (любое) из указанных XPath-выражений (A) *должно* иметь ссылки на некоторые элементы другого выражения (B), обозначаемые одним или несколькими знаками «#» («#» - ссылка на элемент, упомянутый в качестве дочернего по отношению к корню OBJS, «##» - ссылка на элемент, дочерний по отношению к элементу «##»). В проверочном режиме правило вычисляет выражение B, после чего определяет: а) в случае оператора «=>» – существует ли набор узлов, соответствующий выражению A; б) в случае оператора «=>>» – существует ли набор узлов, соответствующий выражению A, причем узлы, соответствующие XPath-выражению-1 должны присутствовать в документе ближе к его началу, чем узлы, соответствующие XPath-выражению-2. Результатом проверки является соответствие (существует) или нестрогое противоречие (не существует). В конструирующем режиме правило достраивает документ, воспринимая XPath-выражение A как конструирующее. Это правило является доминантным, определяющим, фактически, условное достраивание. При этом наличие оператора «=>>» позволяет выполнять достраивание, явно руководствуясь последовательностью «изложения» XML-документа модели, которая в системе PGEN++ совпадает с последовательностью соответствующих элементов в исходной постановке задачи, в ряде случаев отражающей порядок применения каких-либо операций, характерных для процесса решения задачи.

5. Проверочное *правило тестирования*.

[знак] «[» XPath-выражение «]» «:» «{» содержание «}» «.»

содержание = вход_атрибут «,» выход_атрибут [«,» «**NoSpaces**»]

вход_атрибут = идентификатор

выход_атрибут = идентификатор

Правило проверяется только для построенного варианта модели, который успешно прошел проверку по всем остальным правилам. Тогда исполняется указанное XPath-выражение и каждый из тэгов-объектов X результата интерпретируется как тест, который должна пройти программа, порожденная по построенному варианту модели. При этом содержимое атрибута (объекта X) с именем вход_атрибут воспринимается как входная строка I теста, а содержимое атрибута с именем выход_атрибут (того же объекта) воспринимается как выходная строка O теста. Если указан необязательный параметр **NoSpaces**, то строка O сравнивается с консольным выводом порожденной программы без учета пробелов, табуляция и переводов строки. Если же такой параметр не указан, то сравнение производится посимвольно с учетом всех символов.

Замечание о наследовании правил. Если некоторый элемент любого правила записан для объекта-тэга класса A, то этот элемент также справедлив для объекта, относящегося к любому из классов-потомков A (по иерархии классов системы PGEN++ для текущей предметной области).

Отметим, что если в каком-либо из вышеперечисленных правил указана *инверсия*, то его результат R преобразуется в результат U по следующим правилам:

«соответствие» => «строгое противоречие»,

«строгое противоречие» или «нестрогое противоречие» => «соответствие».

В качестве примера приведем составленный вручную общий логический скрипт M_{DIR}, содержащий правила и XPath-функции (используемые в регулярно-логических выражениях при унификации шаблонов T) для учебной предметной области – простой обработки векторных данных (поиск минимума, максимума, среднего):

```
- [/OBJS/clsSimpleTerminator] >> [/OBJS/clsSimpleProgram].
```

```
+ [/OBJS/clsSimpleProgram] >> [/OBJS/clsSimpleBlock].
+ /OBJS/clsSimpleProgram[@ID = "PROG" and (@Name = "PROGRAM" or
true())].
+ [/OBJS/clsSimpleProgram] = [1].
+ [/OBJS/clsSimpleBlock] >> [/OBJS/clsSimpleTerminator].
+ /OBJS/clsSimpleTerminator[@ID = "END"].
+ [/OBJS/clsSimpleTerminator] = [1].
+{3} [/OBJS/clsSimpleMat[@OVar != ""]/O[@ID="Res"]] =>
[/OBJS/clsSimpleScalar[@ID = #/@OVar]/I[@ID="Asgn"]/Link[@Code =
##/@Ref]].
+{3} [/OBJS/clsSimpleScalar[@ID =
#/@IVar]/O[@ID="Handle"]/Link[@Code = ##/@Ref]] =>
[/OBJS/clsSimpleBlock[@IVar != ""]/I[@ID="Arg"]].
+{3} [/OBJS/clsSimpleVector[@ID = #/@IVar and (@Size !=
"")]/O[@ID="Handle"]/Link[@Code = ##/@Ref]] =>
[/OBJS/clsSimpleBlock[@IVar != ""]/I[@ID="Arg"]].
+{6} [/OBJS/clsSimpleBlock[@ID != ""]/O[@ID="Next"]] =>>
[/OBJS/clsSimpleBlock[@ID != "" and @ID !=
#/@ID]/I[@ID="Prev"]/Link[@Code = ##/@Ref]].
+{1.5} [/OBJS/clsSimpleBlock[@ID != ""]/O[@ID="Next"]] =>
[/OBJS/clsSimpleTerminator/I[@ID="End"]/Link[@Code = ##/@Ref]].
+{1.5} [/OBJS/clsSimpleProgram/O[@ID="Begin"]/Link[@Code =
##/@Ref]] => [/OBJS/clsSimpleBlock[@ID != ""]/I[@ID="Prev"]].
+ [/OBJS/clsSimpleBlock] < [10].
+ [/OBJS/clsSimpleProgram[@ID = "PROG" and (@Name = "PROGRAM" or
true())]/O[@ID="Begin"]/Link[@Code = ##/@Ref]] =>
[/OBJS/clsSimpleTerminator[@ID = "END"]/I[@ID="End"]].
+ [/OBJS/clsSimpleTest] :{Input, Output, NoSpaces}.
* action(&$VERB,&$OBJ): /*/Link[Name/text()="MVv" and
   set($VERB,Left/Value/text()) and set($OBJ,Right/Value/text())
or
   Name/text()="MVIv" and set($OBJ,Left/Value/text()) and
   set($VERB,Right/Value/text())].
* obj_attr($OBJ,&$ATTR): (
     count(/*/Link[Name/text()="Mp" and Left/Value/text()=$OBJ
and
     set($JUNCT,Right/Value/text())])>0 or
     count(/*/Link[Name/text()="MIp" and Right/Value/text()=$OBJ
and
     set($JUNCT,Left/Value/text())])>0
   ) and
   count(/*/Link[Name/text()="Jgp" and Left/Value/text()=$JUNCT
and
```

```
   set($ATTR,Right/Value/text())])>0.
* verb_dir($VERB,&$DIR): (
   count(/*/Link[starts-with(Name/text(),"EI") and
     Right/Value/text()=$VERB and
set($JUNCT,Left/Value/text())])>0 or
   count(/*/Link[starts-with(Name/text(),"E") and
Left/Value/text()=$VERB
     and set($JUNCT,Right/Value/text())])>0
  ) and
   count(/*/Link[Name/text()="Jv" and Left/Value/text()=$JUNCT
and
     set($DIR,Right/Value/text())])>0.
```

3.3.3. Распараллеливание работы машины прямого вывода. Ограничения по времени

Базовый алгоритм прямого логического вывода документа ALG является рекурсивным, с порождением множества вариантов на каждом уровне рекурсии, и, как следствие, хорошо распараллеливаемым в системе с общей памятью. Был введен *пул потоков*, количество которых выбиралось в диапазоне N..1,5N, где N – количество имеющихся в системе ядер. На каждом очередном этапе рекурсивного спуска порождаемые варианты Y_i по мере возможности распределялись по свободным потокам. Подмножество вариантов, которые не удавалось «отправить» в параллельные потоки по причине их исчерпания, исполнялись в породившем варианты потоке. Такой подход обеспечил минимальное количество порожденных потоков и, соответственно, минимальные затраты на их обслуживание. Синхронизация потоков выполнялась с помощью семафоров и критических секций. Дополнительно велся контроль множества уже обработанных вариантов (для сокращения объема перебора). Использовалась мемоизация, позволившая обрабатывать каждый уникальный вариант только один раз.

Поскольку алгоритм прямого логического вывода документа-модели управляется весами слабых ограничений (фактически, используется марковская модель), он может показывать различные по затра-

там времени результаты в различных сеансах вывода даже на одной задаче, в зависимости от «удачности» вероятностного вывода по правилам. Соответственно, различие во времени вывода может быть десятикратным и более. По этой причине было принято решение о вводе специального параметра $T_{отс}$ – *максимально допустимого времени сеанса вывода*. После запуска решения задачи машина вывода начинает отсчет времени. Если решение не будет получено за максимально допустимое время, то машина вывода аннулирует текущий сеанс вывода, обнуляет время, проводит реинициализацию и запускает новый сеанс вывода для той же задачи (с аналогичным контролем времени). Как показали эксперименты, такая простая схема достаточно эффективна и позволяет получить результаты за меньшее время (обычно при нескольких перезапусках).

Параллельная машина прямого логического вывода была успешно применена для восполнения постановки задач простой обработки векторных данных. Постановка формулировалась в виде текста на русском языке, система PGEN++ преобразовывала ее в первичную XML-модель (представляющую объектно-событийную модель задачи), которая дополнялась по предложенной в работе методике прямого логического вывода. В таблице 4.1 приведены замеры среднего (по 10 экспериментам в каждом случае) времени работы T, ускорения S и эффективности распараллеливания E, в зависимости от числа ядер N и числа потоков P.

Таблица 4.1. Результаты замеров и характеристики распараллеливания

P	N	T, c	S	E
1	1	16,17	1	1
2	2	7,36	2,2	1,1
4	4	3,74	4,33	1,08
6	4	4,59	3,52	0,88

При P, равном N, показано суперлинейное ускорение, что может объясняться как более эффективным использованием кэш-памяти ядер при разделении задачи на фрагменты, так и случайными факторами,

связанными с вероятностной природой использованного алгоритма вывода в шаге 3.3. Чистая же эффективность распараллеливания, вероятно, близка к единице. Увеличение P сверх N не дало положительных результатов из-за возросших затрат на распараллеливание при том же числе ядер.

3.3.4. Марковская модель управления последовательностью выбора правил

Как уже упоминалось выше, на шаге 3.3 алгоритма работы машины прямого логического вывода имеем набор активных правил (слабых ограничений), сообщивших о нестрогом соответствии и готовых к исправлению соответствующих проблем в текущем варианте документа-модели. Важным вопросом является определение очередности выбора правил.

В данной работе такой выбор осуществляется с применением марковской модели, описываемой, во-первых, *вектором весов правил* $W = (W_1,\ldots,W_k)$, $k = \overline{1,K}$, во-вторых, *матрицей вероятностей переходов между правилами* $M = [M_{kp}]$, $p = \overline{1,K}$, где K – общее число правил, используемых в текущей предметной области.

На каждом шаге 3.3 начинаем с того, что получаем вектор-строку $R = (R_1,\ldots,R_k)$. Если вывод еще только начат, то $\forall k : R_k = 1$, в противном случае $\forall k : R_k = M_{sk}$, где s – номер правила, выбранного на предыдущем шаге вывода. Далее вычисляется ненормированный вектор весов

$$X_k = R_k W_k,$$

который после нормирования становится вектором вероятностей $Y = (Y_1,\ldots,Y_k)$ выбора k-го правила.

Полученный вектор вероятностей Y совместно с генератором равномерно распределенных случайных чисел и используется для определения последовательности применения правил.

Необходимо отметить, что существует *частный случай*, при котором данный вероятностный алгоритм не применяется. Это происходит в случае, если в настройках системы PGEN++ выбрана опция «*Напрямую применять идентичные случаи*» и в базе данных системы хранятся результаты вывода задачи, эквивалентной текущей задаче, включающие трассу вызова правил для успешно полученного результата. В таком случае, при решении текущей задачи правила просто последовательно выбираются, следуя данной трассе.

3.3.5. Поиск оптимальной матрицы переходов с помощью одного варианта обобщенно-регрессионных нейронных сетей

Итак, последовательность перехода между правилами в процессе прямого логического вывода направляется весами правил и матрицей вероятности переходов. Веса правил W задаются пользователем-специалистом в предметной области в скриптах M_{LD}, тогда как определение матрицы переходов M является крайне нетривиальным моментом, который вряд ли может быть максимально успешно выполнен человеком. Соответственно, возникает вопрос о разработке некоего автоматизированного подхода к определению M, например, на базе интерполяции нейронной сетью, использующей собранную историю о маршрутах применения правил (представленную матрицами переходов P0 и P1, см. далее) при решении схожих задач.

Первым, вполне логичным решением, будет выбор матрицы, заполненный равными значениями

$$M_{kp} = \frac{1}{K}$$

при отсутствии какой-либо истории.

Второе решение – хранить в истории для произвольной задачи две матрицы вероятностей переходов – примененную исходную P0 (то

есть P0 = M, причем сохраняется M, рассчитанная для текущего варианта задачи) и *ослабленную* апостериорную P1,

$$\tilde{P} = B \cdot A \cdot B^{T};$$

$$\forall j \forall k : P1_{jk} = \alpha + \beta \frac{\tilde{P}_{jk}}{\max\limits_{s}\left(\tilde{P}_{js}\right)};$$

$$\alpha + \beta = 1,$$

где A и B являются результатом применения алгоритма Баума-Уэлша (по окончании сеанса логического вывода с получением трассы применения правил TR) для нахождения скрытой марковской модели (A, B, π) по имеющимся наблюдениям TR, α и β - эмпирические ослабляющие коэффициенты, препятствующие наличию нулевых и близких к ним вероятностей (в данной работе $\alpha = 0{,}7$; $\beta = 0{,}3$). Вид выражения для $\tilde{P}$ определяется тем, что от наблюдений мы переходим к вероятностям состояний, применяем матрицу переходов между состояниями и, наконец, от состояний снова переходим к наблюдениям.

Выдвинем гипотезу, что при достаточно малом времени, затраченном на вывод решения для текущей задачи, более ценна априорная информация P0 (поскольку решение при такой матрице переходов было быстрым), тогда как при относительно большом времени решения ценнее апостериорная информация P1, которая гарантированно определяет верную последовательность применения правил, в отличие от P0, с которой процесс решения сходился долго.

Проблему нахождения адекватного значения M при наличии истории решения различных (в том числе, возможно, и весьма схожих) задач можно решить путем применения *некоторого варианта обобщенно-регрессионной нейронной сети*, важным достоинством которой является отсутствие затратной по времени процедуры обучения.

Предположим, что *существует история решения задач* прямого логического вывода, каждый i-й элемент которой является четверкой (P0(i), P1(i), C(i), T(i)), где C(i) – некоторый кортеж параметров i-й задачи, а T(i) – время, затраченное на вывод ее решения. Тогда находится матрица

$$M = \frac{\sum_i \omega_i \left[(1-\varphi_i)P0(i) + \varphi_i P1(i)\right]}{\sum_i \omega_i},$$

где ω_i – мера схожести i-й задачи с текущей задачей, а φ_i – коэффициент, учитывающей время решения i-й задачи в соответствии с вышеизложенными рассуждениями, причем

$$\omega_i = \exp\left(-R \cdot dist(C, C(i))^2\right);$$

$$\varphi_i = a + b \cdot \min(1, \gamma \cdot T(i));$$

где R – параметр, определяемый как удвоенное среднее расстояние от кортежа параметров текущей задачи C до некоторого множества (в нашем случае – пяти) ближайших к нему (в пространстве параметров) кортежей задач C(i), dist(C1,C2) – функция, возвращающая некоторую метрику расстояния между C1 и C2 в пространстве параметров, а прочие величины являются эмпирическими константами, в данной работе использовались следующие их значения:

$$a = 0{,}5;\ b = 0{,}25;$$

$$\gamma = 0{,}01.$$

Достаточно существенным вопросом является содержание кортежей C, C(i) и связанный с ним вопрос о вид функции dist(C1,C2). В данной работе *кортеж параметров задачи* содержит вектор ключевых слов (в порядке их появления в постановке задачи), а также числовые матрицы, векторы и скаляры:

а) скалярное время, затраченное системой на решение задачи;

б) вектор частот ключевых слов (ключевыми считаются слова, упомянутые в качестве аргументов и полученные в качестве результатов XPath-функций, являвшихся интерфейсом по отношению к трансформирующему слою [грамматического разбора] и вызывавшихся в процессе работы из регулярно-логических выражений шаблонов);

в) матрицы грамматических отношений между вышеуказанными ключевыми словами (число матриц равняется числу найденных в ходе работы трансформирующего слоя существенных типов отношений).

Функция dist работает по следующему алгоритму:

1. Находит разность d между двумя векторами, каждый из которых содержит все взвешенные элементы соответствующих числовых матриц и векторов, а также и все соответствующие взвешенные скаляры. При этом выбор весов осуществляется, в значительной степени, эмпирически.

2. Находит расстояние Левенштейна между векторами ключевых слов и добавляет его(с некоторым весом) в вектор d.

3. Возвращает эвклидову норму полученного вектора d.

3.4. Построение отчуждаемых версий системы под задачу

Система PGEN++ поддерживает генерацию исполняемых консольных программ, реализующих работу различных распознающих ОСМ (на выбранном множестве индуцирующих классов), принимающих входной текст и формирующих выходной файл (текст или XML-представление выходной ОСМ). При этом сначала генерируется файл на языке Free Pascal, содержащий: а) инициализационные части, наполняющие различные объекты индуктора данными, извлеченными из индуцирующих методов, б) исполняющую (индуцирующую) часть. Полученный файл компилируется стандартными средствами Free Pascal, при этом генерируется исполняемый файл, принимающий параметры через командную строку.

Таким путем могут быть автоматически построены распараллеленные отчуждаемые версии индукторов с простейшим интерфейсом через командную строку (без визуально-графической оболочки). Такие версии работоспособны в операционных системах Windows и Linux на машинах с общей памятью (при произвольном числе ядер).

3.5. Автоматизированное приобретение знаний об индукции ОСМ и их трансформации

Достаточно очевидно, что индуцирующие скрипты четко формализованы, но не вполне просты в составлении. Эти обстоятельства говорят о том, что процедура составления таких скриптов нуждается в автоматизации. Такая автоматизация вполне возможна, по крайней мере, для шаблонированных групп регулярно-логических выражений и для декларативных скриптов прямого логического вывода.

Как будет показано далее, для решения задачи автоматизации вполне достаточно наличия наперед заданных базовых знаний о предметной области (спецификаций классов данной области и правил соединения объектов этих классов в модель задачи) и набора обучающих пар вида «предложение – объекты со связями», где предложение исчерпывающим образом указывает на класс и (в явной или неявной форме) описывает значения полей соответствующего объекта.

Формально опишем каждую обучающую пару как (G, S), где G – объект модели (для которого известны связи f(G)), S – предложение, порождающее данный объект.

3.5.1. Индукция шаблонированных групп регулярно-логических выражений

Выходными данными этого процесса являются шаблоны-группы регулярно-логических выражений, вспомогательные XPath-функции для обеспечения интерфейса со слоем грамматического разбора, CSV-таблицы синонимического или трансляционного (справочного) характера.

Построение шаблонов ведется последовательно, класс за классом. Для каждого класса C в обучающей выборке обнаруживается множество пар (G_C, S), где G_C – один из объектов, относящихся к

классу C. Далее в предложениях S выделяются фрагменты-параметры P (последовательности латинских букв или цифр, являющихся некими параметрами текущего объекта), для которых делается допущение о неизменности порядка их следования в такого рода предложениях. Формируется простой распознающий шаблон в виде группы регулярно-логических выражений, выделяющих очередное предложение и выявляющих требуемое количество фрагментов-параметров.

На основе прямого сопоставления значений полей объектов G_C значениям выделенных параметров P составляются схемы заполнения полей, которые могут включать константы, значения P и/или некие уникальные значения T_i, которые взаимно однозначно сопоставляются обнаруженным в i-ых предложениях характерным словам W_i (для этого составляются CSV-таблицы пар (T_i, W_i), которые формируются с применением простых элиминативно-индукционных методов). Очевидно, что

$$\bigcap_i W_i = \varnothing;$$

$$\bigcap_i T_i = \varnothing.$$

Выделенные схемы заполнения полей записываются в распознающие части индуцирующих скриптов.

Далее, с помощью библиотеки грамматического разбора link-grammar [23] в предложениях S обнаруживаются связки A вида «действие – объект действия» и формируются две синонимические CSV-таблицы – T_1 (варианты глаголов, выражающих действие) и T_2 (варианты существительных, выражающих объект действия). Формируются первые индуцирующие правила, обращающиеся к слою грамматического разбора (с этой целью составляются соответствующие XPath-функции, работающие с виртуальными XML-документами – результатами грамматического разбора), выявляющие связки «действие – объект действия» и проверяющие соответствующие слова-члены связок по таблицам T_1 и T_2. Далее в предложениях S, относящихся к объектам G_C класса C, выявляются уникальные слова U, наличие которых позволяет однозначно говорить о том, что данные предложения дей-

ствительно описывают объекты именно класса C. Если W_i – слова из i-го предложения, относящегося к объектам G_C, а W_j – слова из прочих предложений, то

$$U = \left(\bigcap_i W_i \right) \setminus \left(\bigcup_{j \neq i} W_j \right).$$

Если такие уникальные слова U обнаруживаются, то для каждого из них определяется типовая цепь связок со словами главной связки A «действие - объект действия». Если для некоего U_k такая однозначная цепь связок существует, формируется дополнительное индуцирующее правило, которое проверяет наличие в выделенном предложении соответствующего слова, находящегося в такой характерной цепи связок с элементами связки A. На этом формирование шаблонов завершается и начинается составление правил достраивания прямым логическим выводом, которые будут рассмотрены в следующем пункте.

Предположим, что строится индуцирующий метод для класса «Обработка векторных данных» для предметной области «Простая обработка векторных данных». Пусть обучающая выборка содержит предложения для различных классов, в том числе для данного – два предложения (и, соответственно, два разных объекта, значения полей которых указаны):

(G_{C1}, S_1) = ((IVar = «V»; Op = «Min»; OVar = «min»; ID = «MinVmin»), «Найдем минимум вектора V и поместим результат в скаляр min.»);

(G_{C2}, S_2) = ((IVar = «V»; Op = «Max»; OVar = «max»; ID = «MaxVmax»), «Найдем также максимум вектора V и поместим результат в скаляр max»).

На рис. 3.1 приведен фрагмент построенного шаблона.

```
@global_unique(MAIN,infinity):-
  ()->{V0}()->{V3}()->{VRES0}()->{V4}()->{VRES1}
  (([^\w\.]+(\w+)->{P0}[^\w\.]+(\w+)->{P1}[^\.]*\.)->{Grammar.ru{SENTENCE}} (*PRUNE))?=>{
    xpathf(SENTENCE,'MVv1oi',$V0,'результат','true'),dbconsts1(V0),
    xpathf(SENTENCE,'NXv0io','результат',$V3,'true'),dbtables0(V3,$VRES0),
    xpathf(SENTENCE,'NXv0io','результат',$V4,'true'),dbtables1(V4,$VRES1)
  }.
  @auto:-MAIN:"//P0/text()" => "IVar",MAIN:"//VRES0/text()" => "Op",MAIN:"//P1/text()" =>
"OVar",MAIN:"//VRES0/text()"+MAIN:"//P0/text()"+MAIN:"//VRES1/text()" => "ID".
```

Рис. 3.1. Фрагмент автоматически построенного шаблона

В верхней части данного фрагмента отчетливо обозначены: схемы выделения двух фрагментов-параметров P0 и P1, а также индуцирующие правила, определяющие грамматические отношения уникального для данного класса слова «результат» с элементами главной связки «действие – объект действия» (MVv1oi), а также со словом («минимум» или «максимум»), детерминирующим вид обработки (NXv0io), причем осуществляется трансляция данного слова в значения («Min»/«Max» и «min»/«max») для заполнения полей объекта (по формируемым таблицам dbtables0.csv и dbtables1.csv соответственно). В нижней части фрагмента обозначена схема заполнения полей объекта непосредственно значениями параметров P0 и P1, а также таблично транслированными значениями VRES0 и VRES1.

В качестве примера также приведем текст автоматически сгенерированной XPath-функции NXv0io, возвращающей правое слово (по известному левому) в грамматическом отношении типа «NXv»:

```
* NXv0io($i0,&$o1): (count(/*/Link[Name/text()="NXv" and
Right/Value/text()=$i0 and set($o1,Left/Value/text())]) > 0).
```

3.5.2. Индукция скриптов прямого логического вывода (правил достраивания и трансформации)

Выходными данными этого процесса является набор XPath-подобных правил достраивания. Для их составления используется тот

же набор пар (G_C, S), причем известны связи f(G_C) для каждого из объектов G_C.

Выявляется набор классов, к которым относятся объекты G_C и типы реализованных связей f(G_C) между контактами классов: фактически строится сеть классов (C, E), где каждой дуге – типу связи E = (C_1, C_2) соответствует набор реализованных связей f(G_C) между объектами G_{C1} и G_{C2}. Для каждой дуги E осуществляется попытка создания правила достраивания. В настоящее время строятся только правила восполнения, как наиболее полные и универсальные (в конструирующем режиме они способны достроить как связь между существующими объектами, так и отсутствующие объекты вместе со связями, а в проверочном режиме данные правила позволяют проконтролировать наличие объектов и требуемых связей).

Алгоритм составления каждого правила вполне тривиален, поскольку строятся только те типы связей, которые присутствуют между объектами классов, входящих в обучающую выборку, причем вся необходимая информация о структуре таких связей известна заранее (определена в спецификациях порождающих классов текущей предметной области). Необходимо учитывать лишь следующие обстоятельства:

а) если предполагается возможность создания правилом нового объекта, то необходимо позаботиться о корректном заполнении его полей в конструирующем режиме – для этого проводится анализ текущих экземпляров связи E = (C_1, C_2). Сначала определяется, какой из двух объектов текущего экземпляра связи может быть выведен из значений полей объекта-партнера. У каждого из этих объектов подсчитывается количество M полей, для которых может быть записана общая схема заполнения, состоящая из констант и значений полей объекта-партнера (схема составляется по принципам, схожим с аналогичными принципами для индукции распознающих скриптов). Ведущим (опорным) становится объект с меньшим M (при этом, если у ведомого k-го объекта $M_k < N_k$, где N_k – число полей объекта класса C_k, то правило формируется в проверочно-неконструирующей форме). Если таких

ведущих объектов у связи E определить не удается, то она признается невыводимой и процесс завершается неудачей. Если на роль ведущего претендуют оба объекта, то используется эвристический прием: ведущим становится объект с большей степенью соответствующей вершины в сети классов. Как только определены ведущий объект и схема заполнения полей его объекта-партнера по связи, в правило достраивания объекта-партнера включаются соответствующие конструктивные условия вида «@поле = выражение», где выражение является константным или ссылается на поля парного объекта и другие константы (для соединения частей выражения может использоваться стандартная XPath-функция concat);

б) контакт некоторого типа определяется в одном классе и может наследоваться в его классах-потомках. Это создает некоторые сложности при определении правил. Пусть в экземплярах связи типа $E = (C_1, C_2)$ ведущими являются объекты классов $g(C_1)$, а ведомыми (конструируемыми) являются объекты классов $g(C_2)$, где функция $g(X)$ возвращает множество, включающее класс X и его потомков. Тогда для каждого из классов, входящих в $g(C_2)$, необходимо записать отдельный экземпляр правила с данным конкретным классом. В то же время для ведущего объекта, напротив, целесообразно записать класс общего предка (ближайшего по иерархии) всех $g(C_1)$;

в) веса правил, создающих какую-либо связь типа E, можно определить пропорционально количеству существующих экземпляров такой связи.

Рассмотрим пример из той же предметной области «Простая обработка векторных данных». Пусть обучающая выборка содержит элементы:

(G_{C1}, S_1) = ((IVar = «V»; Op = «min»; OVar = «min»; ID = «minV-min»), «Найдем минимум вектора V и поместим результат в скаляр min.»);

(G_{C3}, S_3) = ((IVar = «V»; ID = «outmin»), «Выведем скаляр min на экран»);

(G_{C4}, S_4) = ((Size = «10»; ID = «V»), «Введем вектор V из 10 элементов»);

(G_{C5}, S_5) = ((ID = «min»), «Введем скаляр min»).

Здесь подразумевается, что если аргументом операции поиска минимума является вектор V, то должен существовать декларирующий его объект, который должен быть связан с операцией поиска минимума. Аналогично, должен существовать объект, декларирующий скаляр min, если данный скаляр выводится на экран.

Можно получить набор правил достраивания, фрагмент которых показан на рис. 3.2.

```
+{6.75} [/OBJS/clsSimpleScalar[@ID != #/@ID and @ID = #/@IVar]/O[@ID="Handle"]/Link[@Code =
##/@Ref]]=>[/OBJS/clsSimpleBlock[@ID != ""]/I[@ID="Arg"]].
+{6.75} [/OBJS/clsSimpleVector[@ID != #/@ID and @ID = #/@IVar]/O[@ID="Handle"]/Link[@Code =
##/@Ref]]=>[/OBJS/clsSimpleBlock[@ID != ""]/I[@ID="Arg"]].
```

Рис. 3.2. Фрагмент полученных правил достраивания

На рисунке показаны два правила, которые утверждают:

1. Необходимость существования объектов (если они не существуют, то будут созданы) класса clsSimpleScalar (скаляр) или clsSimpleVector (вектор), если существует некая обрабатывающая их операция класса clsSimpleBlock с аргументом-контактом Arg. Создаваемые объекты получат идентификаторы ID, равные значениям полей IVar объектов класса clsSimpleBlock.

2. Необходимость существования связи (если она не существует, то будет создана) между контактом Handle декларирующих объектов классов clsSimpleScalar/clsSimpleVector и контактом Arg обрабатывающего объекта класса clsSimpleBlock в случае, если поле IVar обрабатывающего объекта равняется идентификатору ID декларирующего объекта.

Дополнительно отметим, что в данном случае имело место обстоятельство (б) (см. выше), поскольку в обучающей выборке присутствовали связи классов clsSimpleVector=>clsSimpleMat и clsSimpleScalar=>clsSimpleOut, которые формально относятся к единому типу

clsSimpleVar=>clsSimpleBlock. При этом для ведущих объектов классов clsSimpleMat и clsSimpleOut в вышеприведенных правилах был записан их общий класс-предок clsSimpleBlock, а для ведомых были записаны конкретные классы clsSimpleScalar и clsSimpleVector.

3.6. Апробация

В предыдущих пунктах был подробно рассмотрен вопрос о восстановлении полноценной смысловой ОСМ (представленной в формате XML-документа) из исходной, возможно неполной, постановки задачи, а также об автоматизированном построении правил такого восстановления. Рассмотрим теперь некоторые применения данных процессов.

3.6.1. Реконструкция модели. Верификация скриптов, порождающих программу

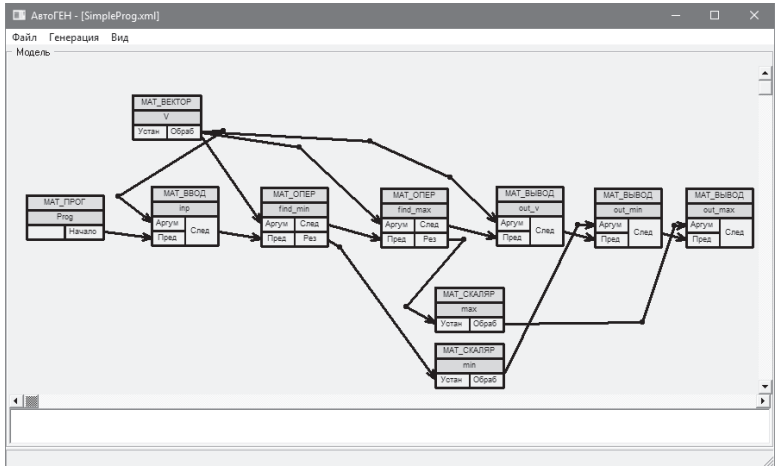
В наиболее простом случае можно непосредственно использовать индуцированную ОСМ для *верификации порождающих программы скриптов*. В самом деле, если была построена некоторая модель A, по которой была порождена программа P, из которой была восстановлена модель B, то при эквивалентности исходной модели A и реконструированной модели B можно говорить о *корректности порождающих методов* модели A (разумеется, при этом мы должны быть уверены в строгой корректности индуцирующих B скриптов).

В настоящее время такая схема верификации *реализована* для скриптов, порождающих учебные программы, решающие элементарные задачи, включающие консольные ввод и вывод, а также математическую обработку (поиск минимума, максимума, вычисление среднего арифметического) векторных данных. Для восполнения первичной модели, полученной в результате унификации шаблонов, исполь-

зуется прямой логический вывод. На рис. 3.4 представлено окно программы PGEN++, иллюстрирующее работу с распознающими классами для подобных задач.

На рис. 3.3 представлены *схемы исходной и реконструированной моделей* одной из таких программ – они изоморфны, следовательно, порождающие скрипты корректны. Приведем текст соответствующей программы:

```
#include <stdio.h>
#include <stdlib.h>
int main() {
  double V[10];
  double max;
  double min;
  int i;
  for (i = 0; i < 10; i++) {
    printf("V[%i] = ", i);
    scanf("%lf", &V[i]);
  }
  min = 1E300;
  for (i = 0; i < 10; i++)
    if (V[i] < min)
       min = V[i];
  max = -1E300;
  for (i = 0; i < 10; i++)
    if (V[i] > max)
       max = V[i];
  printf("V = [");
  for (i = 0; i < 10; i++)
    printf("%lf ", V[i]);
  printf("]\n");
  printf("min = %lf\n", min);
  printf("max = %lf\n", max);
  return 0;
}
```



а)

б)

Рис. 3.3. Схемы исходной (а) и реконструированной (б) моделей программы

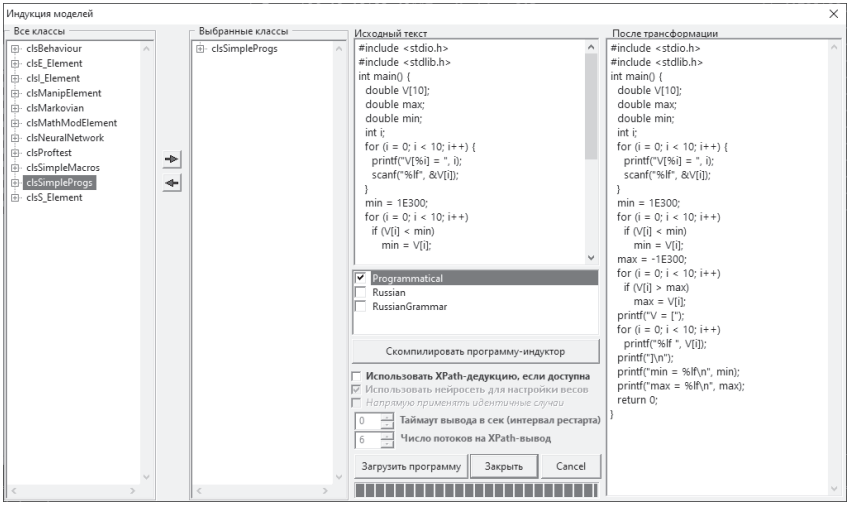


Рис. 3.4. Окно встроенного в PGEN++ индуктора

3.6.1.1. Полуавтоматическая коррекция порождающих скриптов

С задачей верификации порождающих скриптов тесно связана задача их коррекции. В системе PGEN++ существует полуавтоматический режим такой коррекции, когда система выдает текущий результат порождения программы, в который пользователь системы вручную вносит изменения, после чего система пытается *скорректировать, по меньшей мере, константные фрагменты порождающих скриптов*. Подробное рассмотрение механизма такой коррекции выходит за рамки данной работы, опишем лишь общий подход:

1. Порождаемая программа размечается системой, с указанием, для каждого фрагмента, порождающего его класса.

2. Пользователь вносит изменения в порожденную программу, после чего система вызывает стандартную утилиту **diff**, которая генерирует журнал внесенных изменений.

3. Система просматривает журнал изменений и пытается определить константные фрагменты скриптов, соответствующие изменен-

ным фрагментам порожденной программы, и модифицировать их соответствующим образом.

3.6.2. Порождение программ по естественно-языковым описаниям задачи

В настоящее время разработаны распознающие и порождающие классы, позволяющие генерировать решающие программы для *учебных задач обработки векторных данных* (включая консольные ввод и вывод, а также математическую обработку – поиск минимума, максимума, вычисление среднего арифметического) по их описаниям на упрощенном русском языке. При этом используются распознающие методы с двумя версиями разбора – упрощенной (Russian) и основной (RussianGrammar).

Упрощенная версия основана на выделении фрагментов типовых императивных высказываний (без привлечения слоя грамматического разбора) с активным использованием CSV-таблиц синонимов для предикатов типа **nearest**, применение которых позволяет реализовать проверки по словарям с определенной толерантностью к мелким синтаксическим ошибкам. Далее используется обратный логический вывод, восполняющий первичную модель, полученную в результате работы базовой схемы унификации шаблонов: вводятся необходимые отсутствующие объекты-факты (например, постановка могла не содержать фрагмента, соответствующего объекту «Программа в целом»), вводятся факты о наличии необходимых связей. Далее по полученной модели генерируется ее XML-представление, по которому система PGEN++ порождает программу [11].

Такой подход отличается хорошей скоростью работы, но предъявляет достаточно жесткие требования упрощенности к исходным описаниям на русском языке.

Основная версия базируется на выделении регулярно-логическими выражениями отдельных предложений и применении к

ним средств грамматического разбора из трансформирующего слоя, используется XML/XPath-интерфейс с данным слоем, также используются таблицы синонимов из CSV-таблиц. Далее применяется прямой логический вывод, восполняющий первичную модель до полной смысловой.

Данный подход работает несколько медленнее, но при этом он мало чувствителен к перестановкам слов и дополнительным словам и оборотам.

Приведем *пример постановки задачи на естественном языке*:

```
Составить программу. Ввести скаляр max. Ввести скаляр min.
Введем вектор V из 10 элементов. Зададим вектор V с клавиатуры.
Найдем минимум вектора V и поместим результат в скаляр min.
Найдем также максимум вектора V и поместим результат в скаляр
max. Вывести скаляр min на экран. Вывести скаляр max на экран.
Вывести вектор V на экран.
А среднее арифметическое считать не будем.
```


Отметим, что данная постановка может быть сокращена, например, если убрать три первых предложения, то система PGEN++ за счет механизма дополнения смысловой модели восполнит ее в процессе вывода. Аналогично, может быть убрано последнее предложение, не несущее в данном случае смысловой нагрузки и введенное лишь в качестве «лингвистического шума».

На рис. 3.5 приведена *построенная по данной постановке смысловая модель*, а на рис. 3.6 приведена *порожденная по ней программа*:

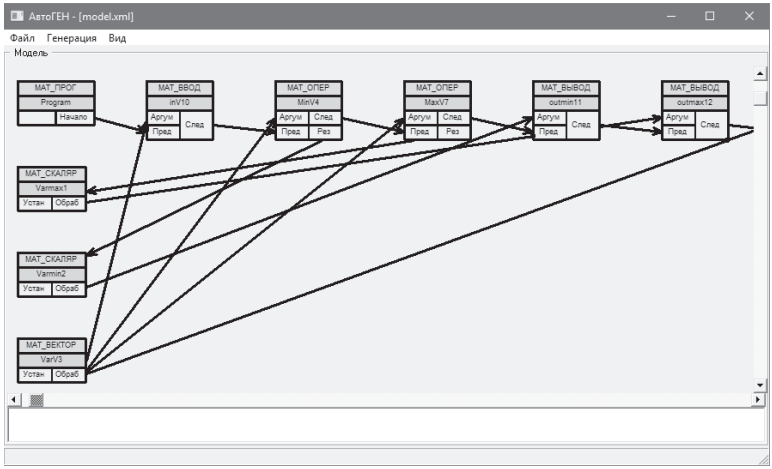


Рис. 3.5. Схема смысловой модели, построенной по естественно-языковому описанию задачи

Рис. 3.6. Окно со сгенерированной системой программой

Дополнительно приведем *вариант постановки той же задачи (с частично измененным порядком слов), с которой справляется только версия с основной схемой (с применением грамматического разбора)*:

```
Введем вектор V из 10 элементов. Вектор V зададим с клавиатуры.
Найдем минимум вектора V, и поместим результат в скаляр min.
Найдем также максимум вектора V и поместим результат в скаляр
max. Вывести на экран скаляр min. Вывести скаляр max на экран.
Вывести вектор V на экран. А среднее арифметическое считать не
будем.
```

3.6.3. Автоматизированное построение шаблонов и правил достраивания

Для предметной области «Простая обработка векторных данных» были написаны несколько текстовых описаний типичных задач, сопоставленных порождающим соответствующие программы моделям. В частности, текстовому описанию задачи из пункта 3.6.2 была сопоставлена модель, показанная на рис. 3.5.

Эти данные использовались для приобретения знаний о трансформации текстовых постановок задач в порождающие модели, способные сгенерировать решающие программы. Такой процесс индукции занял не более 20 секунд, в результате были порождены распознающие скрипты (версии Auto), справочные и транслирующие CSV-таблицы к ним, набор необходимых XPath-функций и правила достраивания.

Результаты здесь полностью не приводятся ввиду большого объема, проиллюстрируем лишь некоторые моменты. Далее приведен фрагмент шаблона для класса «Скалярная величина» (clsSimpleScalar):

```
@versions(Auto)
@fast(dbconsts3,"dbconsts3.csv").
@context(PREV,infinity):-
  (((^)|(\.)+)(\s*\\n)*)\s*.
@glue:-.
@global_unique(MAIN,infinity):-
```

```
   ()->{V0}
   (([^\w\.]+(\w+)->{P0}[^\.]*\.)->{Grammar.ru{SENTENCE}}
(*PRUNE))?=>{
     xpathf(SENTENCE,'MVv1oi',$V0,'скаляр','true'),dbconsts3(V0)
   }.
 @auto:-MAIN:"//P0/text()" => "ID".
 @goal:-
xpath('MAIN','//P0/text()',[VIDText0]),assertz(simplescalar(VIDT
ext0)),!.
 @done:-clear_db.
```

Этот шаблон ссылается на таблицу синонимов dbconsts3.csv, которая содержит две строки:

```
Ввести
Определить
```

Полученный общий скрипт прямого логического вывода приведем полностью (можно сравнить его с аналогичным, но составленным вручную скриптом из пункта 3.3.2):

```
@versions(Auto)
+{1.50} [/OBJS/clsSimpleProgram[@ID != #/@ID and @Name =
"PROGRAM" and @ID = "PROG"]/O[@ID="Begin"]/Link[@Code =
##/@Ref]]=>[/OBJS/clsSimpleInput[@ID != ""]/I[@ID="Prev"]].
+{6.75} [/OBJS/clsSimpleScalar[@ID != #/@ID and @ID =
#/@IVar]/O[@ID="Handle"]/Link[@Code =
##/@Ref]]=>[/OBJS/clsSimpleBlock[@ID != ""]/I[@ID="Arg"]].
+{6.75} [/OBJS/clsSimpleVector[@ID != #/@ID and @ID =
#/@IVar]/O[@ID="Handle"]/Link[@Code =
##/@Ref]]=>[/OBJS/clsSimpleBlock[@ID != ""]/I[@ID="Arg"]].
+{5.25} [/OBJS/clsSimpleInput[@ID !=
#/@ID]/O[@ID="Next"]/Link[@Code =
##/@Ref]]=>[/OBJS/clsSimpleBlock[@ID != ""]/I[@ID="Prev"]].
+{5.25} [/OBJS/clsSimpleMat[@ID !=
#/@ID]/O[@ID="Next"]/Link[@Code =
##/@Ref]]=>[/OBJS/clsSimpleBlock[@ID != ""]/I[@ID="Prev"]].
+{5.25} [/OBJS/clsSimpleOut[@ID !=
#/@ID]/O[@ID="Next"]/Link[@Code =
##/@Ref]]=>[/OBJS/clsSimpleBlock[@ID != ""]/I[@ID="Prev"]].
+{2.25} [/OBJS/clsSimpleMat[@ID !=
""]/O[@ID="Res"]]=>[/OBJS/clsSimpleScalar[@ID != #/@ID and @ID =
#/@OVar]/I[@ID="Asgn"]/Link[@Code = ##/@Ref]].
```

```
+{1.50} [/OBJS/clsSimpleOut[@ID !=
""]/O[@ID="Next"]]=>[/OBJS/clsSimpleTerminator[@ID != #/@ID and
@ID = "END"]/I[@ID="End"]/Link[@Code = ##/@Ref]].

* Mp1ii($i0,$i1): (count(/*/Link[Name/text()="Mp" and
Left/Value/text()=$i0 and Right/Value/text()=$i1]) > 0).

* MVIv1ii($i0,$i1): (count(/*/Link[Name/text()="MVIv" and
Left/Value/text()=$i0 and Right/Value/text()=$i1]) > 0).

* MVv1ii($i0,$i1): (count(/*/Link[Name/text()="MVv" and
Left/Value/text()=$i0 and Right/Value/text()=$i1]) > 0).

* MVv1io($i0,&$o1): (count(/*/Link[Name/text()="MVv" and
Left/Value/text()=$i0 and set($o1,Right/Value/text())]) > 0).

* MVv1oi(&$o0,$i1): (count(/*/Link[Name/text()="MVv" and
set($o0,Left/Value/text()) and Right/Value/text()=$i1]) > 0).

* NXv0io($i0,&$o1): (count(/*/Link[Name/text()="NXv" and
Right/Value/text()=$i0 and set($o1,Left/Value/text())]) > 0).
```


Было показано, что полученные скрипты/таблицы/функции/правила могут успешно использоваться для генерации решающих программ по текстовым постановкам задач на русском языке. Проверка проводилась как по постановкам, использовавшимся при обучении системы, так и по модифицированным. Были получены абсолютно корректные результаты, однако следует отметить, что автоматически порожденные правила работали несколько медленнее аналогичных, составленных вручную. Это объясняется тем, что автоматически составленные правила не содержали правил восполнения с отношением строгого следования «=>>» (в настоящее время автоматическое составление таких правил не поддерживается), которые позволяют уменьшить размерность пространства состояний и существенно ускорить процесс генерации программы по описанию.

3.6.4. Подходы к трансформации программ на базе обратного логического вывода

Выше была подробно рассмотрена проблема трансформации ОСМ на базе прямого логического вывода. Также были обозначены основные подходы к применению обратного логического вывода для решения таких задач. В данном пункте будет подробно рассмотрена более частная задача – трансформация программ. Здесь возможны, по крайней мере, два основных подхода: прямая трансформация и опосредованная трансформация.

3.6.4.1. Прямая трансформация

Прямая трансформация наиболее пригодна для простых текстовых преобразований. Здесь полная переработка входного текста осуществляется в рамках единственного распознающего метода (единственного распознающего класса), включающего лишь шаблон T и скрипт обратного логического вывода M_L. При этом первичная ОСМ не формируется, скрипт обращается напрямую (с помощью XPath-запросов) к результатам разбора, помещенным в элементарные переменные (см. пункт 2.1.3.1) и представленным в виде деревьев переменных/значений, и применяет элементарные модификации входного текста напрямую.

Для доступа к результатам разбора исходной программы, представленным в виде дерева, используется предикат **xpath**, описанный в пункте 3.2. Что же касается элементарных модификаций, то они представлены предикатами:

insert_before(EXPR, ID, вставляемый_текст) – вставляет текст в результат разбора, опосредованный строковым идентификатором **EXPR** одного из-регулярно-логических выражений, перед разобранным элементом, соответствующим узлу дерева переменных/значений с идентификатором **ID**);

insert_after(EXPR, ID, вставляемый_текст) – аналогичен **insert_before**, но вставка производится после указанного элемента;

change(EXPR, ID, замещающий_текст) – аналогичен предыдущим предикатам, но указанный текст полностью заменяет текущий текст указанного элемента.

В качестве примера приведем полностью распознающий (индуцирующий) метод класса **clsDePythonize**, который обрабатывает текст с Python-подобной системой отступов для обозначения блоков программы и преобразует ее в C-подобную систему обозначения блоков программы, вставляя необходимое количество фигурных скобок:

```
@unique(LINE,infinity):-
  (\s*\\n)->{EMPTY}|(^(\s*)->{SB}([^\\]*)->{LB}\\n)|((\s*)
->{S}([^\\]*)->{L}\\n)->{FULL}|($)->{END}.
pop_braces(M,ID,[H|T],H,[H|T]):-
  insert_before(M,ID,H),
  insert_before(M,ID,'}\\n'),
  !.
pop_braces(M,ID,[H|T],V,T1):-
  insert_before(M,ID,H),
  insert_before(M,ID,'}\\n'),
  pop_braces(M,ID,T,V,T1),
  !.
handle:-
  xpath('LINE','/EMPTY',[_]),
  !.
handle:-
  xpath('LINE','/SB[text()='''']',[_]),
  asserta(offs([''])),
  !.
handle:-
  xpath('LINE','/SB[text()!='''']/text()',[SBText]),
  xpath('LINE','/SB[text()!='''']/SB',[SBID]),
  asserta(offs([SBText,''])),
  insert_before('LINE',SBID,'{\\n'),
  !.
handle:-
  xpath('LINE','/FULL/S/text()',[SText]),
  offs([SText|_]),
  !.
handle:-
  xpath('LINE','/FULL/S/text()',[SText]),
  offs([SPrev|T]),
  atom_length(SPrev,N1),
```

```
  atom_length(SText,N2),
  <(N1,N2),
  xpath('LINE','/FULL/S',[SID]),
  insert_before('LINE',SID,SPrev),
  insert_before('LINE',SID,'{\\n'),
  retractall(offs(_)),
  asserta(offs([SText,SPrev|T])),
  !.
handle:-
  xpath('LINE','/FULL/S/text()',[SText]),
  offs([SPrev|T]),
  atom_length(SPrev,N1),
  atom_length(SText,N2),
  >(N1,N2),
  xpath('LINE','/FULL',[FID]),
  pop_braces('LINE',FID,T,SText,T1),
  retractall(offs(_)),
  asserta(offs(T1)),
  !.
handle:-
  xpath('LINE','/END',[EID]),
  offs([SPrev|T]),
  atom_length(SPrev,N1),
  >(N1,0),
  pop_braces('LINE',EID,T,'',_),
  retractall(offs(_)),!.
handle:-
  xpath('LINE','/END',[_]),
  offs(['',_]),
  retractall(offs(_)),!.
@goal:-
  handle.
@done:-
  clear_db.
```


3.6.4.2. Опосредованная трансформация

Данный подход является существенно более мощным и универсальным по сравнению с прямой односкриптовой трансформацией. Здесь также используются только шаблоны T и скрипты обратного логического вывода M_L, но распознавание уже ведется не единствен-

ным классом, а системой классов. Трансформация проходит в три этапа:

1. Распознавание входной программы с формированием классами первичной цепочечной ОСМ, возможно сопровождаемой вспомогательными информационными структурами (например, записями в CSV-таблицах), хранящими, фактически, значения полей объектов такой ОСМ.

2. Обработка полученной первичной ОСМ – трансформация, предполагающая вставку/удаление/замену некоторых объектов. Применение на данном этапе логического программирования обеспечивает возможность решения таких достаточно интеллектуальных задач, как автоматическое распараллеливание программ.

3. Формирование выходного текста программы – может выполняться как путем непосредственной генерации выходного трансформированного текста скриптами M_L, так и путем генерации выходной ОСМ, по которой система PGEN++ способна породить выходную программу в обычном режиме.

В настоящее время в рамках опосредованной трансформации реализованы два основных приложения PGEN++: автоматический распараллеливатель С-программ с применением директив Cilk++ [7, 8] и автоматический распараллеливатель С-программ на базе сверхоптимистичных вычислений [13] в частично транзакционной памяти.

В работе [7] приведены фрагменты индуцирующих скриптов, фрагменты скрипта обратного логического вывода класса, осуществляющего Cilk++-распараллеливание С-программ, а также пример распараллеленной программы.

Работа по автоматическому распараллеливанию с применением сверхоптимистичных вычислений на данный момент еще не опубликована, подробное изложение соответствующих подходов выходит за рамки данной работы. Приведем только примеры исходной (Приложение П5) и автоматически распараллеленной программы (Приложение П6) моделирования динамики пучка заряженных частиц

в электростатической линзе, полученной в результате применения построенного на базе опосредованной трансформации параллелизатора.

Выводы к третьей главе

Предложена схема оперативной обработки/трансформации построенных первичных линейных ОСМ в логических скриптах, позволяющая реализовать интеллектуальные алгоритмы на базе средств обратного логического вывода языка GNU Prolog и/или на базе разработанной машины прямого логического вывода. Сформулированы синтаксические средства описания индуцирующих (восстанавливающих и обрабатывающих ОСМ) скриптов.

Предложен новый параллельный алгоритм работы машины прямого логического вывода на базе системы слабых ограничений (правил). Показано, что распараллеливание является оправданным. Для управления процессом вывода предложено использовать новый подход, состоящий в применении марковской модели, матрица переходов которой генерируется на основании анализа схожих случаев. Генерация матрицы выполняется с применением обобщенно-регрессионных сетей с неклассическим вычислением весовых коэффициентов, при этом тексты на естественном языке формально представляются в виде группы векторных, матричных и скалярных параметров. Обоснована необходимость интерполяции между двумя базовыми матрицами переходов (исходной и результирующей для каждого прецедента).

Предложенные схемы, подходы и алгоритмы апробированы на задаче верификации порождающих программу скриптов и на задаче генерации решающей программы по естественно-языковому описанию исходной проблемы. Показано, что применение трансформирующего слоя [грамматического разбора] позволило не только упростить распознающие шаблоны, но и повысить качество распознавания исходных естественно-языковых описаний.

В рамках формализма объектно-событийных моделей впервые предложен механизм автоматизированной генерации шаблонов, распознающих первичные объекты моделей, описаны основные моменты их получения. Также впервые предложен механизм генерации логических правил – слабых ограничений. Важным позитивным отличием данных решений является полное отсутствие необходимости в ручном составлении какой-либо части правил преобразования постановок в XML(DSL)-модели программ. Вручную составляются только правила генерации решающих программ по DSL-моделям. Предложенные решения успешно апробированы на предметной области «Простая обработка векторных данных», позитивные результаты получены как на обучающей выборке, так и на примерах, не входящих в выборку.

Описаны основные подходы к трансформации программ на базе обратного логического вывода. Приведены примеры применения трансформации, в том числе автоматического распараллеливания С-программ в стиле Cilk++ или с использованием сверхоптимистичных вычислений в частично транзакционной памяти.

Заключение

Все задачи, поставленные в работе, выполнены. Получены следующие *основные результаты*:

1. Дано понятие объектно-событийной модели, описаны ее структура и правила интерпретации, позволяющие порождать программу по исходной постановке задачи, данной в произвольной форме (при условии, что она однозначно отображается в порождающую ОСМ). Приведена основная информация, необходимая для написания порождащих методов.

2. Предложены распознающие объектно-событийные модели. Предложены два способа интерпретации распознающих ОСМ, в результате которой генерируются первичные смысловые модели для поставленной задачи. Используется новая концепция шаблонов как переборных групп из контекстных и основных модифицированных регулярно-логических выражений, имеющих простой и доступный интерфейс (в виде специальных предикатов) с элементарными CSV-базами данных (БД), с трансформирующим слоем грамматического разбора и, в некоторых случаях, с глубокими нейронными сетями прямого распространения.

3. На базе синтаксиса XPath предложен простой алгоритмический микроязык, для которого показана алгоритмическая полнота. Введено понятие конструирующего XPath-запроса, являющегося одним из средств данного микроязыка, определены синтаксис и семантика предложенных алгоритмических конструкций. Предложена схема применения микроязыка в регулярно-логических выражениях при оперативной логико-алгоритмической обработке первичных результатов разбора входного текста.

4. Предложена схема оперативной обработки/трансформации первичных линейных ОСМ в логических скриптах прямого и/или обратного вывода, позволяющая реализовать интеллектуальные алгоритмы вывода выходных полноценных смысловых моделей (ОСМ)

решения задачи, по которым системой PGEN++ могут быть порождены решающие программы. Сформулированы синтаксические средства описания индуцирующих скриптов.

5. Предложен новый параллельный алгоритм работы машины прямого логического вывода на базе системы слабых XPath-подобных ограничений (правил). Для управления процессом вывода предложено использовать марковскую модель, матрица переходов которой генерируется на основании анализа схожих задач. Для генерации матрицы переходов предложено использовать обобщенно-регрессионные сети с неклассическим вычислением выходной функции [с интерполяцией между двумя базовыми матрицами переходов (исходной и результирующей для каждого прецедента].

6. Вышеупомянутые схемы, подходы и алгоритмы апробированы на задаче верификации порождающих программу скриптов и на задаче генерации решающей программы по естественно-языковому описанию исходной задачи. Показано, что применение трансформирующего слоя [грамматического разбора] позволило не только упростить распознающие шаблоны, но и повысить качество распознавания исходных описаний.

7. В систему PGEN++ добавлена возможность автоматического построения распараллеленных отчуждаемых версий индукторов, генерирующих выходные смысловые модели, с простейшим интерфейсом через командную строку (без визуально-графической оболочки). Такие версии работоспособны в операционных системах Windows и Linux на машинах с общей памятью (при произвольном числе ядер). Данная возможность позволяет производить трудозатратный вывод конечных смысловых моделей на мощных вычислительных системах, работающих под управлением Linux.

8. В рамках формализма объектно-событийных моделей впервые предложен механизм автоматизированной генерации шаблонов, распознающих первичные объекты моделей, и логических правил – слабых ограничений. Важным позитивным отличием данных решений является полное отсутствие необходимости в ручном составлении ка-

кой-либо части правил преобразования естественно-языковых постановок в XML(DSL)-модели программ. Предложенные решения успешно апробированы на предметной области «Простая обработка векторных данных», позитивные результаты получены как на обучающей выборке, так и на примерах, не входящих в выборку.

9. Описаны основные подходы к трансформации программ на базе обратного логического вывода. Приведены примеры применения трансформации, в том числе автоматического распаралелливания С-программ в стиле Cilk++ или с использованием сверхоптимистичных вычислений в частично транзакционной памяти (данная задача решена впервые).

Автор выражает благодарность своим коллегам, принимавшим активное участие в проработке некоторых идей.

В настоящее время текущая полная версия системы переведена в категорию Freeware и доступна для скачивания с личного сайта автора по ссылке

http://pekunov.byethost31.com/Progs.htm#PGEN

Библиографический список

1. Воеводин В.В., Воеводин Вл.В. Параллельные вычисления. — СПб.: БХВ-Петербург, 2002. — 608 с.

2. Дюк В., Самойленко А. Data mining: учебный курс.— СПб: Питер, 2001. — 368 с.

3. Зубков В. П., Назаретский С. П. IPGS — интеллектуальная система автоматизированного программирования// Инф. среда вуза: Сб. ст. Иваново: ИГАСА, 2000. С.213-215.

4. Ильин В.Д. Система порождения программ. М.: Наука, 1989. - 264 с.

5. Паронджанов, В. Д. Язык Дракон. Краткое описание.- М., 2009.- 124 с.

6. Пекунов В.В. Автоматизация параллельного программирования при моделировании многофазных сред. Оптимальное распараллеливание // Автоматика и телемеханика.— 2008.— №7. — С.170-180.

7. Пекунов В.В. Автоматическое распараллеливание С-программ в Cilk++ стиле. Применение индукции объектно-событийных моделей. - LAP LAMBERT Academic Publishing, 2018. - 105 с.

8. Пекунов В.В. Автоматическое распараллеливание С-программ с применением директив Cilk++ на базе распознающих объектно-событийных моделей // Программные системы и вычислительные методы. — 2018. - № 4. - С.124-133. DOI: 10.7256/2454-0714.2018.4.28086. URL: http://e-notabene.ru/ppsvm/article_28086.html

9. Пекунов В.В. Искусственные нейронные сети прямого распространения. Описание с помощью расширенных машин Тьюринга, вербализация и применение в аэродинамике. - LAP LAMBERT Academic Publishing, 2016. - 177 с.

10. Пекунов В.В. Метаслой моделирования алгоритма, данных и функциональных характеристик последовательных и параллельных программ // Информационные технологии.- 2011. - №6. - С.51-56.

11. Пекунов, В.В. Новые методы параллельного моделирования распространения загрязнений в окрестности промышленных и муниципальных объектов // Дис. докт. тех. наук. — Иваново, 2009. — 274 с.

12. Пекунов В.В. Объектно-событийные модели порождения программ // Вестник ИГЭУ. - Иваново, 2004. - Вып.3. - С.49-52.

13. Пекунов В.В. Сверхоптимистичные вычисления: концепция и апробация в задаче о моделировании электростатической линзы // Программные системы и вычислительные методы. — 2020. - № 2. - С.37-44. DOI: 10.7256/2454-0714.2020.2.32232. URL: https://e-notabene.ru/ppsvm/article_32232.html

14. Пекунов, В.В. Теория объектно-событийных моделей последовательных и параллельных программ // Тр.межд.конф. "Современные телекоммуникационные системы и компьютерные сети: перспективы развития". - СПб.:СПбГАСУ, 2011. - С.231-246.

15. Рассел С., Норвиг П. Искусственный интеллект: современный подход.— М.: «Вильямс», 2007. — 1408 с.

16. Уоссермен Ф. Нейрокомпьютерная техника: Теория и практика.— М.: Мир, 1992.

17. Чарнецки К., Айзенекер У. Порождающее программирование: методы, инструменты, применение.— СПб.: Питер, 2005.— 731 с.

18. Шуин, С.В. Язык описания алгоритмов в системе ИСАП / С.В. Шуин, А.А. Аленкин, В.П. Зубков // Тез. докл. Междунар. науч.-техн. конф. «Состояние и перспективы развития электротехнологий» (XI Бенардосовские чтения). — Иваново, 2003. — Т.1. — С.56.

19. Языковый инструментарий: новая жизнь языков предметной области (Domain Specific Languages) / Фаулер М. // Технология клиент-сервер. — 2005. — № 3. — С. 3-14.

20. Boyle J.M., Harmer T.J., Winter V.L. The TAMPR Program Transformation System: Simplifying the Development of Numerical Software / Arge E., Bruaset A.M., Langtangen H.P. (eds.) Modern Software Tools in Scientific Computing. — Birkhäuser, 1997. — P.353-372.

21. Clark S., Curran J.R. 2007. Widecoverage efficient statistical parsing with CCG and log-linear models // Computational Linguistics 33(4). pp. 493–552.

22. Gvero T., Kuncak V. (2015). Interactive Synthesis Using Free-Form Queries // 37th IEEE International Conference on Software Engineering (ICSE). 689-692 pp. DOI: 10.1109/ICSE.2015.224.

23. Link Grammar Parser. URL: https://www.abisource.com/projects/link-grammar/

24. Mandal S., Naskar S. Natural Language Programing with Automatic Code Generation towards Solving Addition-Subtraction Word Problems // Proceedings of 14th International Conference on Natural Language Processing (December, 2017). Jadavpur University, 2017. 146-154 pp.

25. Oda Y., Fudaba H., Neubig G. et al. Learning to Generate Pseudo-code from Source Code using Statistical Machine Translation // 30th IEEE/ACM International Conference on Automated Software Engineering (ASE), 2015. pp. 574-584.

26. Perlre: [https://perldoc.perl.org/perlre.html].

27. Press, W.H. et al. Numerical recipes in C: The art of scientific computing. — Cambridge University Press, 1992.— 994 p.

28. Raza M., Gulwani S., MilicFrayling N. Compositional program synthesis from natural language and examples // Proceedings of IJCAI, 2015. 792-800 pp.

29. Taherkhani, A. Automatic Algorithm Recognition Based on Programming Schemas and Beacons: A Supervised Machine Learning Classification Approach // Doctoral Dissertation.- Aalto University, 2013.- 254 pp.

30. Wong E., Yang J., and Tan L., Autocomment: Mining question and answer sites for automatic comment generation // Proc. ASE, 2013, pp. 562–567.

31. Yin P., Neubig G. A syntactic neural model for general-purpose code generation. In: ACL (1), pp. 440–450 (2017).

Приложения

Приложение П1. Подготовка к запуску программы

В настоящее время программа PGEN++ существует в версии для операционной системы **Windows**. Тем не менее, генерируемые с ее помощью отчуждаемые программы-индукторы работоспособны и в **Linux**, для их нормального функционирования необходимо лишь предварительно установить **PHP** версии **4.3.0** и **GNU Prolog 1.4.4**.

Основной функционал программы готов к применению сразу после распаковки каталога проекта со всеми файлами и подкаталогами, каких-либо существенных настроек проводить не требуется.

Однако *если предполагается запуск порожденной программы из интерфейса системы*, то необходимо для каждого языка **P** установить соответствующий компилятор, а также создать и настроить файлы с именами **start_P.bat** и **start_P.sh**, принимающие от двух до трех параметров. Первый параметр – либо имя компилируемого файла, либо константа **nocompile**, указывающая, что файл уже скомпилирован (имя программы - **_.exe**). Второй параметр – имя выходного файла. Третий (необязательный) параметр – имя файла – входного потока. Программа должна выводить код результата запуска в файл **_.result**. Приведем пример файла для языка **C** (**start_C.bat**):


```
@echo off
@set PATH="C:\Program Files\Microsoft Visual Studio
12.0\VC\bin";"C:\Program Files\Microsoft Visual Studio
12.0\Common7\IDE\";%PATH%
@set IPATH="C:\Program Files\Microsoft Visual Studio
12.0\VC\include"
@set IPATHSDK="C:\Program Files\Microsoft
SDKs\Windows\v7.1A\Include"
@set LPATH="C:\Program Files\Microsoft Visual Studio
12.0\VC\lib"
@set LPATHSDK="C:\Program Files\Microsoft
SDKs\Windows\v7.1A\Lib"
@if "%1"=="nocompile" goto m1
@copy /Y %1 _.c
@cl.exe /O2 -o _.exe _.c /MTd /EHsc /I%IPATH% /I%IPATHSDK% /link
/LIBPATH:%LPATH% /LIBPATH:%LPATHSDK% >_.err
```

```
:m1
@if "%3"=="" goto noin
@_.exe >%2 <%3
@goto end
:noin
@_.exe >%2
:end
echo %errorlevel% >_.result
```

Файл **start_C.sh** выглядит несколько более просто:

```
#!/bin/bash

if [[ "$1" != "nocompile" ]] ; then
  cp -f ./$1 ./_.c
  gcc -O2 -o _.exe _.c >_.err
fi;
if [[ "$3" != "" ]] ; then
  ./_.exe >$2 <$3
else
  ./_.exe >$2
fi;
echo $? >_.result
```


Заметим, что вышеописанный механизм также может быть использован для произвольной обработки порожденного текста. Достаточно в порождающих PHP-скриптах применить вызов функции **PlanAutoStart("название_обработчика")** и создать файлы **start_название_обработчика.bat** и **start_название_обработчика.sh**. В частности, именно таким образом реализован вызов подсистемы **nnets_simplify**, производящей вербализацию и упрощение обученной нейронной сети для каталога классов **clsNeuralNetwork**.

Аналогично, если предполагается компиляция программ-индукторов (в окне, описанном в приложении [Приложение П3]), то дополнительно необходимо установить компилятор **Free Pascal** и настроить его вызов – указать каталог нахождения файла **fpc.exe** в файле **run_fpc.bat**, например:

```
@echo off
d:\lazarus\fpc\3.0.2\bin\i386-win32\fpc -B -O3 -WC -Mobjfpc -
FcUTF-8 %1.pas > %2
@cls
```

Данный файл принимает два параметра: базовое имя компилируемой программы (без расширения) и имя выходного файла.

Приложение П2. Руководство по работе в основных окнах программы

После загрузки программы на экране появляются главные окна системы (Рис. П2.1). Опишем интерфейс этих окон и его смысловую нагрузку.

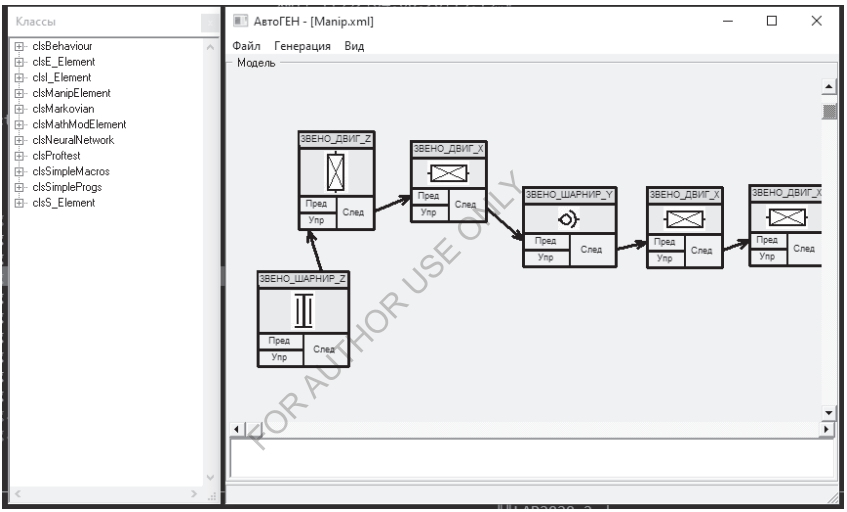


Рис. П2.1. Главные окна системы

Правое окно содержит визуальный редактор моделей, левое – иерархию классов, содержащихся в системе. Окно визуального редактора содержит верхнюю полоску меню, центральное поле редактирования и нижний блок, предназначенный для вывода служебных сообщений (преимущественно о подключении текущего экземпляра системы к группе экземпляров на других машинах в локальной сети).

Процесс создания и редактирования моделей

Готовая модель может быть загружена в окно редактирования с помощью верхнего меню (**Файл/Открыть**). Для создания новой модели достаточно активировать пункт меню **Файл/Новый**. Созданная модель может быть сохранена на диске (меню **Файл/Сохранить** или **Файл/Сохранить как…**). При использовании режима **Файл/Сохранить как…** можно указать формат файла моделей, в настоящее время поддерживаются два формата: а) XML-формат, который является предпочтительным; б) AMD-формат, позволяющий сохранить модель в бинарной форме – этот формат рекомендуется использовать лишь для совместимости со старыми версиями системы.

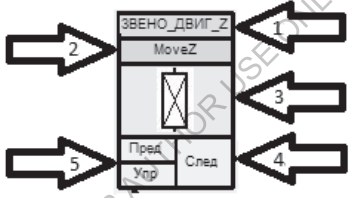


Рис. П2.2. Отображение объекта

Модель создается и редактируется в центральном поле окна редактора в визуальной форме, как граф, вершинами которого являются объекты, а связями – отношения между контактами объектов. Отображение вершины (объекта) содержит пять частей, показанных на рисунке (Рис. П2.2):

1. Имя класса.
2. Идентификатор объекта.
3. Иконка класса (содержится в файле **image.jpg** в папке класса, может отсутствовать).
4. Блок выходных контактов.
5. Блок входных контактов.

Первые три части являются необязательными и могут быть показаны или скрыты в зависимости от настроек объекта – при создании нового объекта правила его отображения берутся из системных настроек программы, при редактировании уже имеющегося объекта его настройки можно посмотреть и изменить в системном окне (Рис. П2.3), вызываемом из контекстного меню объекта (**Контекстное меню/Отображение**). Необходимо отметить, что если объект относится не к обычному классу, а является контейнером-подмоделью, то он будет отображаться более темным цветом по сравнению с объектом, показанным на рисунке выше.

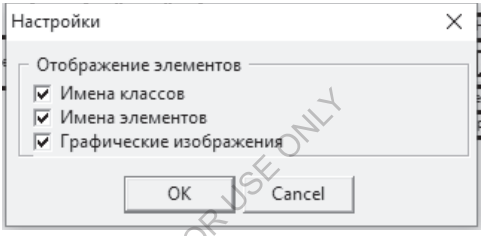


Рис. П2.3. Окно настроек отображения объекта

Вставка, редактирование и удаление объектов

Для создания нового объекта достаточно найти элемент с названием его класса в окне с иерархией классов (если это окно невидимо, то его можно включить через пункт верхнего меню **Вид/Классы**) и с помощью мыши перетащить элемент в поле редактирования модели.

Удаление объекта может быть произведено через контекстное меню (**Контекстное меню/Удалить**). При удалении объекта также удаляются все его связи.

Объекты модели могут быть перемещены мышью. При перемещении объекта также изменяют положение концевые сегменты его связей с другими объектами.

Каждый объект модели имеет набор свойств (полей), определяемых его классом. Этот набор содержит, как минимум его идентификатор, который должен быть уникальным в пределах текущей мо-

дели. Остальные свойства, как уже было сказано, определяются видом класса и его декларацией (содержится в фале **Class.ini** папки класса). При создании нового объекта его свойства получают значения по умолчанию и должны быть отредактированы.

Для вызова редактора свойств необходимо или дважды щелкнуть мышью по объекту или применить контекстное меню (**Контекстное меню/Параметры**).

Если класс не является контейнером-подмоделью, то свойства соответствующего объекта включают уникальные и унаследованные свойства класса. Пример окна свойств объекта обычного класса показан на рисунке (Рис. П2.4).

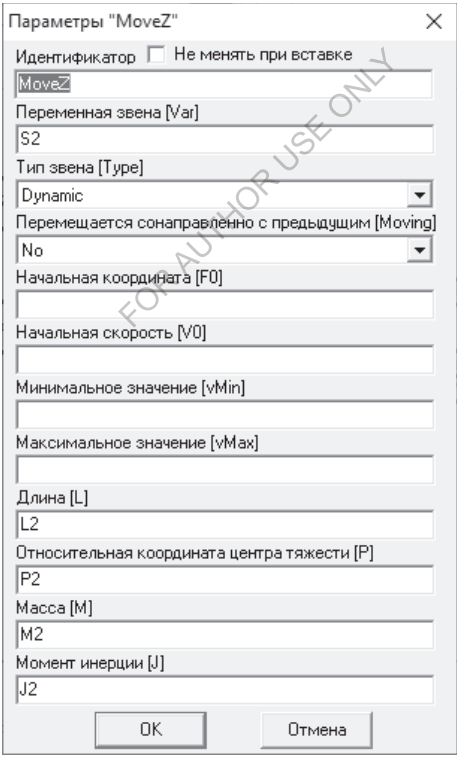


Рис. П2.4. Пример окна просмотра и редактирования свойств объекта обычного класса

В целом, данное окно не требует каких-либо пояснений, за исключением флажка «Не менять при вставке». Флажок имеет смысл только в том случае, если проектируется не самостоятельная модель, а конетейнер-подмодель. Если этот флажок выставлен, то это означает, что данный объект гарантированно не изменит свой идентификатор K при вставке текущей подмодели в какую-либо модель. Если же флажок не выставлен, то при вставке данной подмодели в другую модель под именем X, данный объект будет переименован в X_K.

Пример окна свойств объекта-подмодели показан на рисунке (Рис. П2.5).

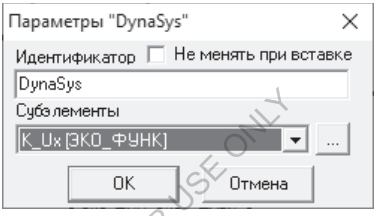


Рис. П2.5. Пример окна просмотра и редактирования свойств объекта-подмодели

Данное окно содержит поле ввода идентификатора и список объектов, входящих в соответствующий контейнер-подмодель. Для редактирования свойств какого-либо из таких объектов достаточно выбрать его в выпадающем списке и нажать расположенную справа от списка кнопку редактирования (**…**). При этом появляется окно свойств подчиненного элемента (Рис. П2.6), в котором идентификатор объекта содержит имя родительского объекта-подмодели и недоступен для редактирования.

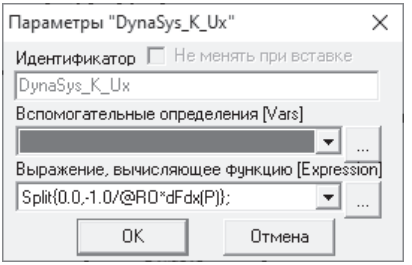


Рис. П2.6. Пример окна просмотра и редактирования свойств объекта, входящего в контейнер-подмодель DynaSys

Соединение объектов связями

Объекты модели могут быть связаны основными или вспомогательными (информационными) связями. По умолчанию создается основная связь, которую можно превратить в информационную через контекстное меню связи (**Контекстное меню/Тип**). Основные связи отображаются толстыми линиями, вспомогательные – тонкими. Циклы по основным связям не допускаются, поэтому цикл возможен только в том случае, если в него входит по крайней мере одна вспомогательная связь.

Для создания связи достаточно щелкнуть мышью по выходному контакту какого-либо объекта, который при этом подсвечивается синим цветом (кроме того, при этом красным цветов подсвечиваются все возможные для соединения входные контакты объектов, Рис. П2.7), и далее, либо щелкнуть мышью по допустимому входному контакту какого-либо объекта [при этом будет создана простая линейная (односегментная связь)], либо еще раз щелкнуть мышью по начальному выходному контакту [для отмены проведения связи].

Удаление связи можно произвести через контекстное меню (**Контекстное меню/Удалить**).

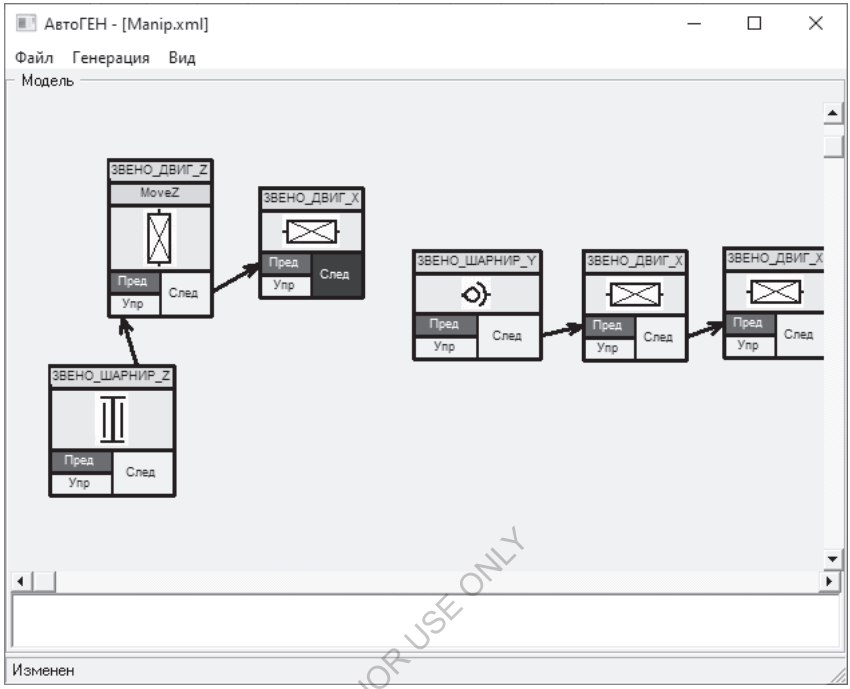


Рис. П2.7. Пример окна редактирования модели в режиме проведения связи с подсветкой контактов

Созданная связь может быть отредактирована:

1. **Изменение цвета**. Доступно через контекстное меню (**Контекстное меню/Цвет**).

2. **Добавление промежуточных точек**. Достаточно «потянуть и переместить» мышью произвольную точку произвольного линейного участка связи.

3. **Перемещение промежуточных точек**. Может быть осуществлено мышью.

4. **Удаление промежуточных точек**. Промежуточные точки удаляются *только автоматически*. Если возникла необходимость удаления некоторой точки, то необходимо мышью переместить ее таким образом, чтобы два сегмента связи, которые она соединяла, при-

близительно встали в одну линию – тогда система удалит данную промежуточную точку.

Создание контейнеров-подмоделей

Контейнеры-подмодели являются специальными классами, представляющими группы связанных объектов. Чтобы создать подмодель (с идентификатором класса М), необходимо:

а) создать требуемую группу объектов,

б) установить в ней все связи,

в) опубликовать контакты тех объектов, которые будут входными или выходными для подмодели,

г) создать папку класса М в дереве каталогов, представляющих классы,

д) создать в папке класса специальный файл описания **Class.ini**,

е) сохранить созданную подмодель в папке класса.

В данном случае файл описания должен содержать пустую секцию [**Parameters**], а также заполненную секцию [**Definition**], содержащую имя класса (в параметре **Name**) и ссылку на созданный файл подмодели (в параметре **Reference**).

Пример файла **Class.ini** для подмодели dynPU.xml:

```
[Definition]
Name=ЭКО_ДИН_СКОР_ДАВЛ
Reference=dynPU.xml
[Parameters]
```

Публикация контактов

Для публикации входного или выходного контакта необходимо вызвать контекстное меню для контакта и выбрать пункт **Опубликовать контакт**. Появится окно (Рис. П2.8), в котором указывается, следует ли публиковать контакт, а также внешнее имя (данное имя будет отображаться как имя контакта объекта-подмодели) и внешний иден-

тификатор (под которым данный контакт будет фигурировать в системе).

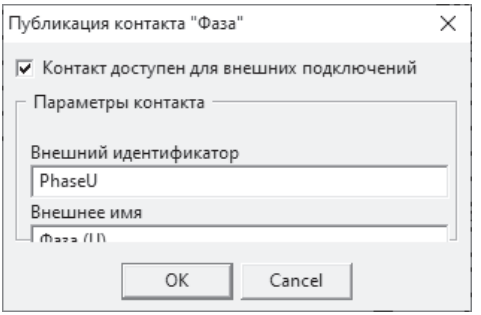


Рис. П2.8. Окно публикации контакта

Опубликованные контакты помечаются зеленым (входные) или красным (выходные) цветным треугольником. Пример фрагмента созданной подмодели показан на рисунке (Рис. П2.9).

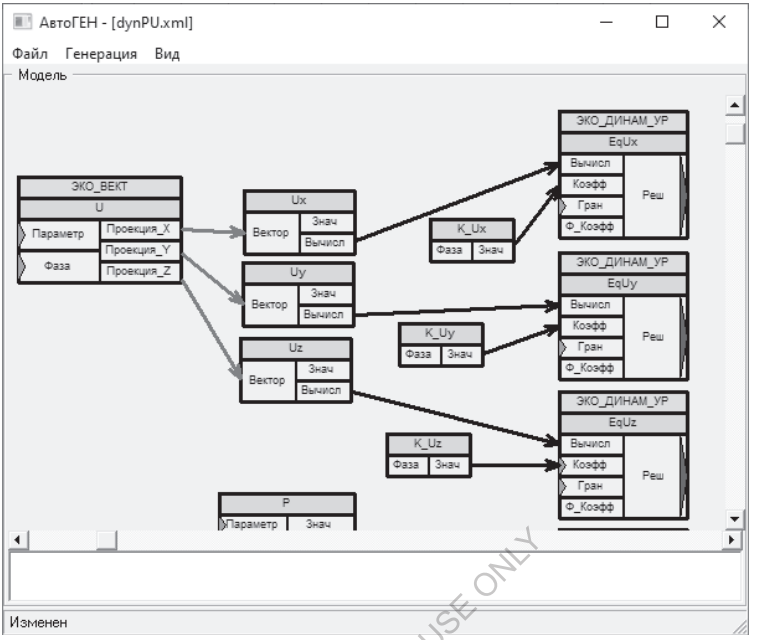


Рис. П2.9. Пример фрагмента созданной подмодели

Генерация (порождение) программы

После создания или загрузки модели можно приступить к генерации (и, возможно, к исполнению) программы.

Если модель генерирует код на обобщенном алгоритмическом языке, то предварительно необходимо определить, на какой из языков высокого уровня будет переведен данный код (сейчас поддерживаются **Pascal** и **C**). Это можно сделать через верхнее меню (**Генерация/Язык конкретизации**), выбрав один из указанных там языков, или выбрав пункт **Нет**, если требуется выходной код именно на обобщенном языке. Набор языков конкретизации можно расширить, создав трансляционный файл на языке **SNOBOL4** и отредактировав файл **Automodelling.ini**: в секцию **[Languages]** вписывается дополнительный параметр, именем которого является название языка конкретиза-

ции, а значением – путь (относительно каталога программы) и имя трансляционного **SNOBOL**-файла. Данная стратегия является достаточно устаревшей и подробно описываться не будет, в случае возникновения интереса читатели могут самостоятельно проанализировать трансляционные файлы **auto_c.sno** и **auto_pas.sno**, находящиеся в каталоге программы.

Генерация программы инициируется через верхнее меню (**Генерация/Транслировать**). При этом, возможно, будет выдан запрос на сохранение файла, после чего система приступит к проверке формальной правильности модели по связям (контакты, маркированные как обязательные, должны иметь связи; контакты, допускающие только одну связь, должны иметь не более одной связи).

В случае нарушения требований к связям, система выдаст информационное сообщение и вернется в режим редактирования. Если же проверка завершилась успешно, то система попытается обнаружить в модели решающие (дедуктивные) блоки.

Если дедуктивные блоки присутствуют, то система собирает соответствующие классам этих блоков файлы **init.pl** и **solve.pl** (находятся в папках классов) и формирует из них дедуктивный скрипт, в который включается консультационный вызов библиотеки предикатов **autoutil.pl**. Данный скрипт исполняется системой GNU Prolog, в результате чего возникает модифицированная модель, которая снова отправляется на генерацию (она также может содержать или не содержать объекты решающих классов).

Если дедуктивные блоки отсутствуют, то система собирает скрипты **script.php3** из папок порождающих классов и формирует из них генерационный файл на языке PHP (подключает **autoutil.php3** и **XPath.class.php**), соответствующий модели. Файл запускается.

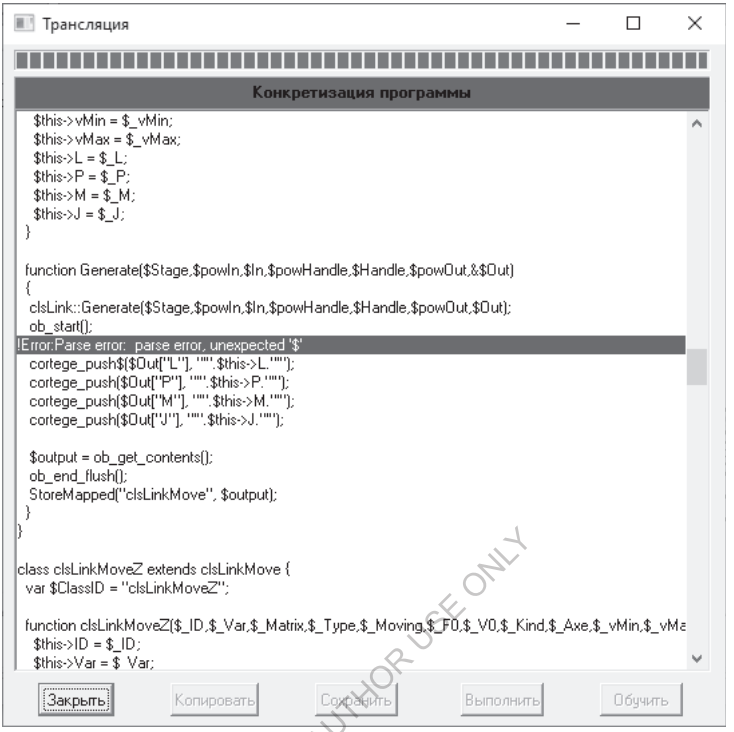


Рис. П2.10. Пример окна с сообщением об ошибке в генерационном файле

Если в процессе запуска интерпретатор PHP выявит синтаксические ошибки, ошибки модели (генерируемые порождающими скриптами в случае неких логических ошибок) или ошибки исполнения, то будет выдано специальное окно с PHP-кодом генерационного файла, в котором перед строкой с возникшей ошибкой находится особая строка с красным фоном, содержащая описание возникшей ошибки. Пример такого окна показан на рисунке (Рис. П2.10). Очевидно, что в приведенном случае ошибка чисто синтаксическая.

Если в процессе генерации не происходит ошибок, то возникает порожденный текст. Как уже упоминалось выше, если он написан на обобщенном алгоритмическом языке, а в настройках указана конкретизация на один из подключенных языков, то полученный текст будет

пропущен через соответствующй транслятор, при этом будет получен новый порожденный текст на требуемом языке.

Рис. П2.11. Пример окна с порожденным текстом

Порожденный текст обычно является текстом программы, но вполне возможна ситуация, когда это какое-либо текстовое описание (например, для системы порождающих классов, описывающих нейронные сети, это может быть текст арифметических вербализирующих выражений, аппроксимирующих полученную обученную нейронную сеть). В любом случае результат выводится в специальное окно, показанное на рисунке (Рис. П2.11).

Данное окно позволяет отредактировать полученный текст и, кроме того, содержит набор кнопок, с помощью которых можно осуществить следующие полезные действия:

1. **Копирование полученного текста в буфер обмена**. Достаточно нажать кнопку **Копировать**.

2. **Компиляция и выполнение полученного текста**. Это возможно, если порождающие скрипты осуществили вызов функции планирования автостарта **PlanAutoStart** с указанием языка. Достаточно нажать кнопку «**Выполнить**». При этом будет вызван компилирующий командный файл, который сгенерирует и запустит полученный код. При этом текстовый вывод, произведенный данным кодом в процессе исполнения, будет в помещен в поле, содержавшее текст программы.

3. **Простейшее приобретение знаний о порождении**. Имеется в виду небольшая автоматическая коррекция константных частей (помещенных между тегами **?>** и **<?**) порождающих PHP-скриптов. Достаточно отредактировать в поле полученный текст и нажать кнопку **Обучить**. Система попытается найти PHP-скрипты, генерировавшие измененный текст, и отредактировать соответствующие константные части. В случае возникновения проблем система выдает специальное окно (Рис. П2.12) с текстом порождающего скрипта, в котором проблемные строки (если их выявить удалось) будут подсвечены красным цветом, и предложит отредактировать его вручную (данное окно содержит кнопку **Редактор** для переключения между режимами отображения ошибочных строк и редактирования скрипта). Поскольку в таком случае приобретение знаний останавливается, настоятельно рекомендуется, после внесения правок в скрипт, снова нажать конпку **Обучить** и повторять указанные операции вплоть до устранения всех ошибок.

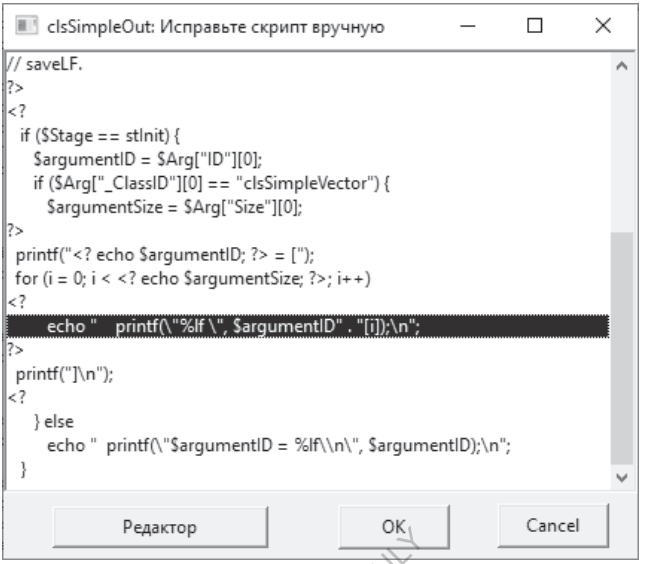


Рис. П2.12. Пример окна с предложением исправить скрипт вручную

Настройки программы

Окно настроек программы вызывается из верхнего меню (**Вид/Настройки**). Вид окна показан на рисунке (Рис. П2.13).

Настройки делятся на две большие группы:

1. **Отображение элементов**. Это настройки, определяющие, какие части графического элемента – объекта модели должны отображаться. Если отмечен пункт «**Использовать для новых элементов**», то эти настройки действуют только для создаваемых новых объектов (напомним, что для конкретного объекта отображаемые элементы всегда можно настроить индивидуально, через **Контекстное меню/Отображение**). Если же отмечен пункт «**Применить ко всем элементам**», то указанные настройки будут принудительно применены ко всем уже существующим в модели объектам.

2. **Автозапуск**. Обычно, при генерации выходного текста/программы, система запрашивает подтверждение на выполнение

операций начала дедуктивного вывода модели (при наличии дедуктивных/решающих классов) и компиляции/запуска сгенерированной программы. Выставив отметки в соответствующих пунктах, можно избежать появления таких запросов – указанные операции всегда будут выполняться. Пункт «**Продолжать вывод модели при порождении фактов**» в настоящее время не задействован [сейчас, если в процессе работы дедуктивный скрипт порождает новые глобальные факты (вызовом специального предиката **plan_asserta**), то дедукция будет продолжена в любом случае, даже если модель уже не содержит решающих объектов].

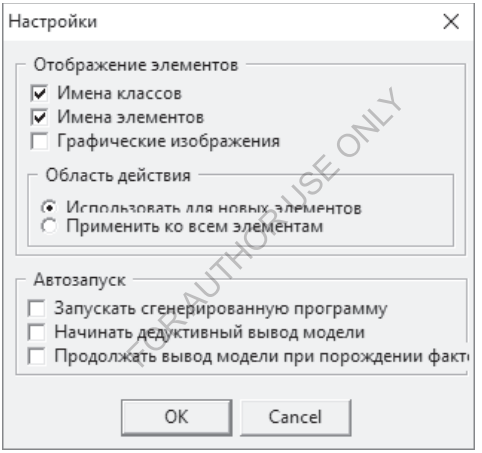


Рис. П2.13. Вид окна настроек

Приложение П3. Руководство по работе в окне индукции моделей

В окне индукции моделей, вызываемом из верхнего меню **Файл/Индукция моделей**, можно осуществлять реконструкцию и преобразование программ/текстов. При этом вид выполняемой операции полностью определяется файлом настроек **induct.ini**, находящимся в корневой папки применяемой иерархии классов, а также файлами – индуцирующими скриптами **induct.mac**, находящимися в папках классов. Файл **induct.ini** может содержать следующие переменные:

а) **Result**. В эту переменную помещается имя файла, в который должны записываться результаты процесса. Если значение не указано, то результат выводится только в окно;

б) **Order**. Содержит перечень имен индуцирующих классов через запятую, указанных в том порядке, в котором они должны применяться ко входному тексту;

в) **Continuous**. Логический признак (0/1), указывающий, какую схему индукции применить: по произвольному тексту (0) или по программе (1), см. раздел 2.1.2. По умолчанию имеет значение (0);

г) **Versions**. Содержит список версий индуцирующего скрипта, перечисленных через запятую, см. раздел 2.1.6.1. По умолчанию пуст. Этот список выводится в окне индукции моделей при выборе иерархии классов с данным **induct.ini**;

д) **Transformer**. Содержит указание трансформирующей DLL-библиотеки и режима трансформации по умолчанию (см. раздел 2.1.3.2). Данные указания разделяются точкой.

Пример файла **induct.ini** для иерархии классов простой обработки векторных данных:

```
Order=clsSimpleProgram,clsSimpleScalar,clsSimpleVector,clsS
impleMat,clsSimpleInput,clsSimpleOut,clsSimpleTerminator,cl
sSimpleTest
Transformer=Grammar.ru
```

```
Versions=Russian,Programmatical,RussianGrammar,Auto
Result=model.xml
```

Вид окна индукции, соответствующего выбору вышеуказанной иерархии, показан на рисунке (Рис. П3.1).

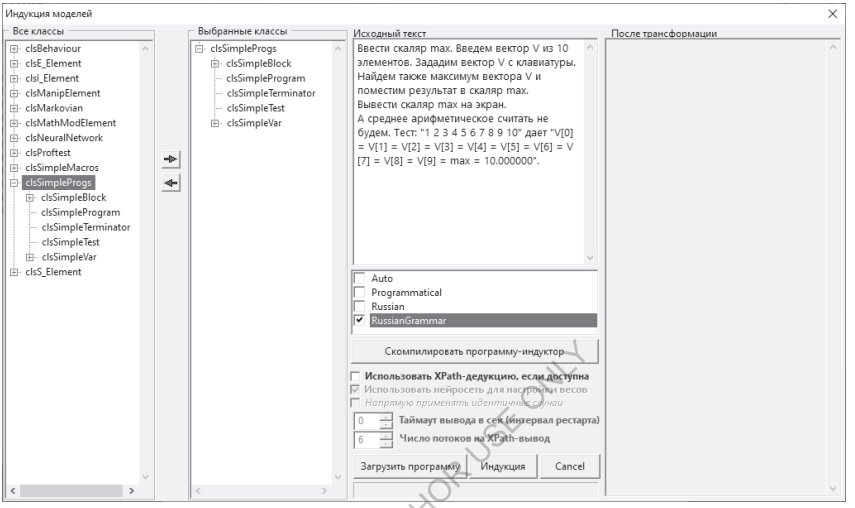


Рис. П3.1. Вид окна индукции моделей

В правой части окна находится список всех классов системы. Далее идет поле **Выбранные классы**, которое должно содержать иерархию тех классов, которые будут использованы в текущем сеансе индукции. Можно либо перетаскивать классы между полями **Все классы** и **Выбранные классы** с помощью мыши, либо воспользоваться кнопками с красными стрелками, указывающими направление перемещения. Перетаскивать необходимо только корневые элементы соответствующих иерархий, подчиненные им элементы будут перемещены автоматически.

После выбора иерархии классов, в поле в центральной части окна может появиться список вариантов (версий) индуцирующих скриптов (если он предусмотрен разработчиком выбранной иерархии классов). В таком случае необходимо выбрать с помощью мыши рабочую версию скриптов (может быть выбран только один вариант).

Исходный текст для индукции помещается в поле **Исходный текст** – данное поле доступно для редактирования, можно пользоваться вставкой из буфера обмена. Кроме того, можно загрузить текст в данное поле из файла – для этого достаточно нажать кнопку **Загрузить программу**.

Дальнейшие действия могут быть осуществлены по одному из четырех стандартных сценариев.

Индукция (обратный логический вывод)

Опирается на логические скрипты, содержащиеся в файлах **induct.mac** из каталогов классов выбранной иерархии. Для этого снимается пометка с флажка **Использовать XPath-дедукцию, если доступна**, после чего необходимо нажать на кнопку **Индукция**. В процессе индукции в поле **После трансформации** будут отображаться информационные сообщения. В случае возникновения ошибок будут выданы соответствующие всплывающие окна с краткой информацией, при отсутствии ошибок в поле **После трансформации** будет помещен полученный трансформированный текст (в зависимости от семантики индуцирующих скриптов это будет либо требуемый результат трансформации текста[1] [Рис. П3.2], либо исходный текст), а в файл, указанный в переменной **Result** файла **induct.ini** будет помещена модель - результат индукции (если такой результат предусмотрен семантикой индуцирующих скриптов). Эта же модель будет загружена в визуальный редактор системы[2], можно ее посмотреть, закрыв окно индукции. В частности на рисунке [Рис. П3.3] показана полученная в результате

[1] Это возможно, например, при работе с иерархией clsDePythonize, которая преобразует текст программы, отформатированный в стиле языка Python (с отступами) в текст, оформленный по стандартам языка C (с фигурными скобками).

[2] Система не проверяет, был ли результирующий файл создан в текущем сеансе или ранее, поэтому в случае ошибок процесса индукции в редактор может загрузиться файл-модель, оставшийся с предыдущей успешной индукции.

индукции модель, соответствующая запуску, изображенному на предыдущем рисунке [Рис. П3.1].

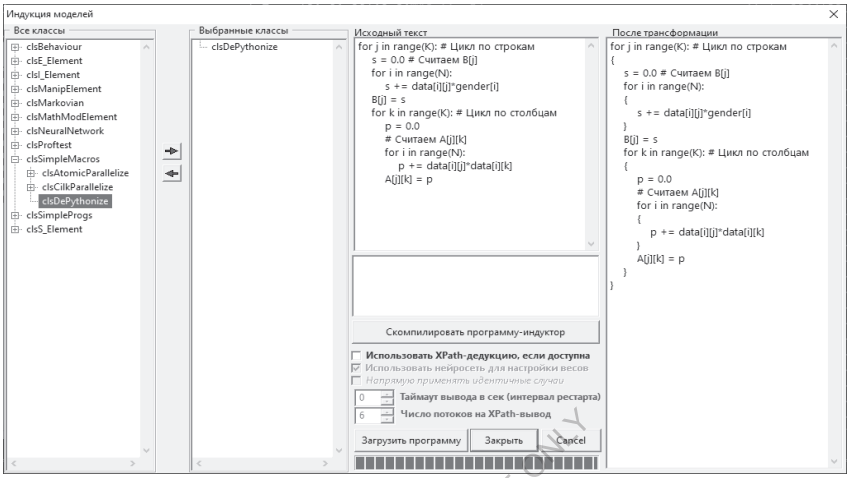


Рис. П3.2. Вид окна индукции с результатом преобразования текста

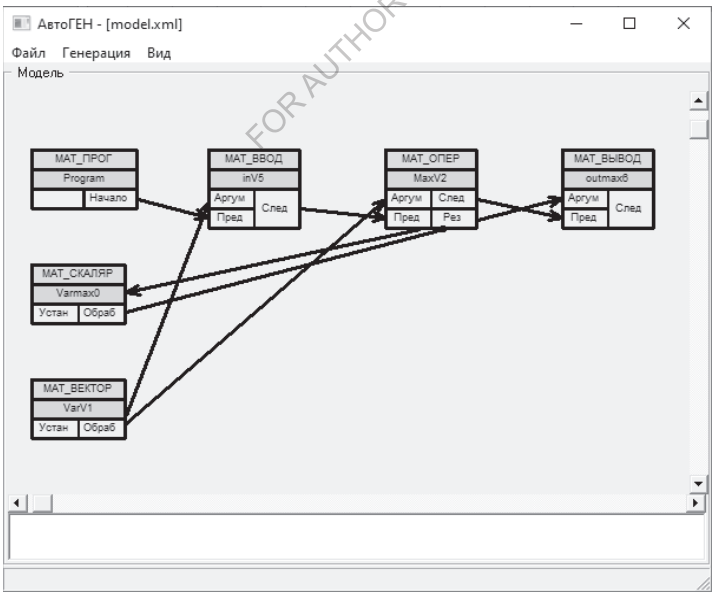


Рис. П3.3. Вид окна редактора с полученной моделью

Индукция (прямой логический вывод)

Опирается на логический скрипт, содержащийся в файле **induct.im** из каталога корневого класса выбранной иерархии. Для этого ставится пометка на флажке **Использовать XPath-дедукцию, если доступна**, после чего необходимо настроить процессинговые параметры индукции и нажать на кнопку **Индукция**.

Процессинговыми параметрами являются:

- **Использовать нейросеть для настройки весов**. При выборе данного флажка оптимальная матрица переходов марковской модели (см. раздел 3.3.5) будет определяться по истории предыдущих сеансов вывода с применением обобщенно-регрессионной нейронной сети. Как правило, такой выбор улучшает показатели вывода и уменьшает требуемое на индукцию время;

- **Напрямую применять идентичные случаи**. При выборе данного флажка в случае, если текущая задача идентична одной из решавшихся ранее, к ней будет применена та же последовательность правил, которая использовалась в предыдущих сеансах. Выбор такого флажка несколько уменьшает потенциальный объем статистики вывода (что слабо негативно отражается на последующих сеансах индукции), но может существенно снизить время индукции в идентичных случаях;

- **Таймаут вывода в сек**. Это максимальное время на непрерывный сеанс индукции, по истечении которого он сбрасывается и автоматически инициируется новый сеанс с новыми случайными параметрами. Поскольку для управления индукцией используется марковская модель, данный параметр весьма полезен для оперативной отбраковки сеансов вывода, оказавшихся чрезмерно далеко от цели. Рекомендуется выставить его в пределах 30÷60 секунд. При указании нулевого значения рестарт в ходе индукции не производится;

- **Число потоков на XPath-вывод**. Прямой логический вывод – чрезвычайно трудоемкая операция, поэтому настоятельно рекомендуется выставлять в данной настройке приблизительно полуторное чис-

ло ядер процессора. Это позволит распараллелить прямой логический вывод и существенно ускорить данный процесс.

В случае возникновения ошибок будут выданы соответствующие всплывающие окна с краткой информацией, при отсутствии ошибок, по окончании процесса в поле **После трансформации** будет помещен исходный текст, а в файл, указанный в переменной **Result** файла **induct.ini** будет помещена модель - результат индукции. Эта же модель будет загружена в визуальный редактор системы.

Создание отчуждаемой версии программы-индуктора

Фактически, речь идет об автоматической генерации и компиляции программы, осуществляющей индукцию по выбранной иерархии классов в выбранном режиме (прямого или обратного логического вывода). Для этого производятся соответствующие настройки процесса индукции (см. пункты выше), однако далее применяется не кнопка **Индукция**, а кнопка **Скомпилировать программу-индуктор**. При этом будет создан исходный файл **_.pas**, для которого будет вызван стандартный компилятор **Free Pascal**. Будет запрошено имя создаваемого исполняемого файла, сообщения компилятора будут выведены в поле **После трансформации**. Вид окна индукции моделей после выполнения данной операции показан на рисунке (Рис. П3.4).

Заметим, что индуктор вполне может быть скомпилирован и в операционной системе **Linux** – для этого достаточно перенести в нее папку с системой, включая созданный файл **_.pas** (а также прочие **pas/inc**-файлы системы), и скомпилировать его стандартным компилятором **fpc**.

Скомпилированный индуктор использует различные файлы из каталога системы (**.bat/.sh**, **.pl**, **.php3**, **.exe**), работает через интерфейс командной строки. При запуске индуктора без параметров, он выводит краткую справку о своих параметрах. Обычно они включают указание входного и выходного файлов (первый и второй параметры), а в

случае применения прямого логического вывода к ним добавляются необязательные: величина таймаута и число используемых потоков.

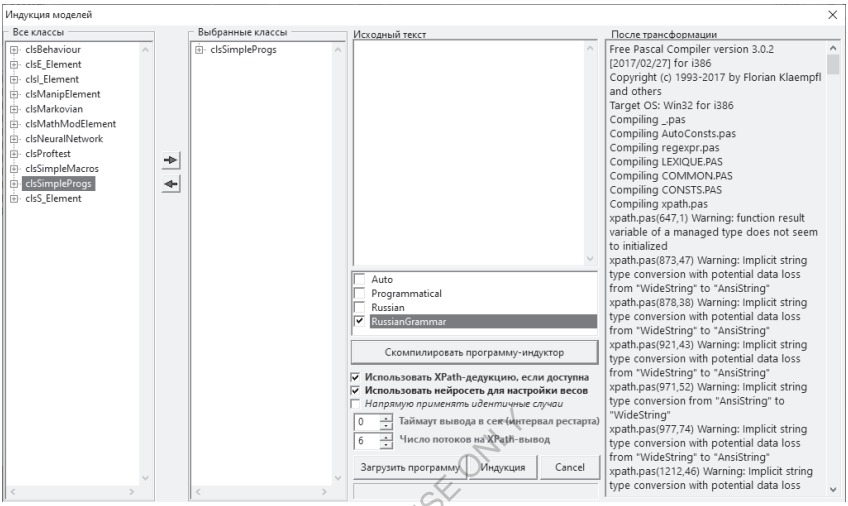


Рис. П3.4. Вид окна после генерации программы-индуктора

Выход из окна индукции моделей

Достаточно нажать кнопку **Cancel** или просто закрыть окно.

Приложение П4. Руководство по работе в окне приобретения знаний

В окне приобретения знаний, вызываемом из верхнего меню **Файл/Индукция правил**, можно осуществлять автоматизированное построение индуцирующих методов M_I, а именно: индивидуальных логических скриптов обратного вывода (M_L), индивидуальных шаблонов-групп (T) из модифицированных регулярно-логических выражений, а также общего логического скрипта прямого вывода (M_{DIR}).

Необходимо заметить, что подсистема приобретения знаний в PGEN++ *ориентирована, преимущественно, на прямой логический вывод* и лишь его полностью автоматизирует. Что же касается обратного логического вывода, то здесь можно говорить лишь о частичной автоматизации, поскольку генерируемые скрипты M_L лишь генерируют базу фактов о распознанных первичных объектах модели. Окончательное же построение модели (на основе этих фактов) все же должно осуществляться неким обобщающим скриптом обратного логического вывода, создаваемым пользователем-программистом системы *вручную*. Обычно это скрипт, связанный с классом, вызываемым в ходе индукции последним (то есть его идентификатор является последним в цепочке идентификаторов, определенной в переменной **Order** файла **induct.ini**).

Вид окна приобретения знаний показан на рисунке [Рис. П4.1].

При запуске окна автоматически создается новый проект, однако можно загрузить уже существующий проект с помощью кнопки **Загрузить проект**. Текущий проект в любой момент может быть сохранен с помощью кнопки **Сохранить проект**.

Проект включает набор готовых моделей, отображаемых в поле **Модели**, набор сопоставлений объектов моделей описывающим их фразам на русском языке (отображаются в поле **Сопоставления**), а также указания базовых имен для генерируемых таблиц (отображают-

ся в полях **Базовое имя справочных таблиц** и **Базовое имя трансформирующих таблиц**).

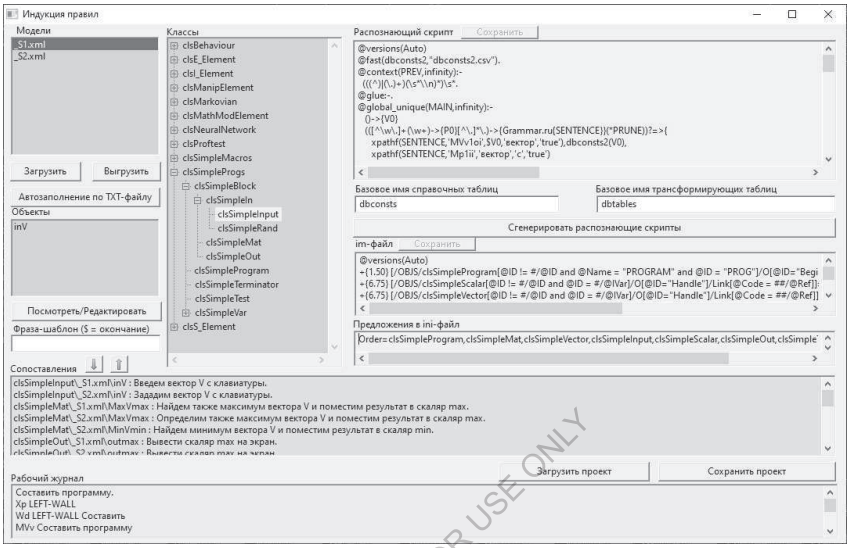


Рис. П4.1. Вид окна приобретения знаний

Создание и редактирование проекта

Разработка проекта заключается в определении набора сопоставлений объектов различных моделей фразам на русском языке, по которым эти объекты можно однозначно индуцировать. Этот набор должен быть непротиворечив и достаточен для восстановления индуцирующих правил.

Для загрузки моделей следует воспользоваться кнопкой **Загрузить**. Загруженная модель появляется в списке в поле **Модели**. Для удаления модели из проекта достаточно выделить ее в списке и нажать кнопку **Выгрузить**.

Сопоставление объектов фразам может осуществляться либо в автоматическом, либо в ручном режиме.

Автоматическое сопоставление подразумевает, что существует некоторая модель A, включающая набор объектов, и текстовый файл B, включающий набор предложений на русском языке (число предложений должно строго равняться числу объектов), причем внутренний порядок следования объектов (его проще всего определить, просмотрев XML-файл модели A) соответствует порядку следования описывающих их предложений. Это существенное ограничение, поэтому такой режим обычно используется, если имеется текстовый файл B, полученный в результате запрограммированного экспорта модели A[1]. Для активизации автоматического сопоставления необходимо загрузить модель в поле **Модели**, затем нажать кнопку **Автозаполнение по TXT-файлу**. При этом будет запрошено имя соответствующего описывающего текстового файла. Если процесс автосопоставления проходит успешно, набор полученных сопоставлений будет отображен в поле **Сопоставления**. В случае неуспеха будут выданы соответствующие диагностические сообщения.

Для *ручного сопоставления* некоторого объекта фразе необходимо:

1. Выделить имя модели в поле **Модели**.

2. Найти и выделить имя класса объекта в поле **Классы**. При этом список всех объектов такого класса, имеющихся в выделенной модели, появится в списке поля **Объекты**.

3. Найти и выделить имя необходимого объекта в поле **Объекты**. Если при этом необходимо изменить какие-либо свойства данного объекта, можно воспользоваться кнопкой **Посмотреть / Редактиро-**

[1] В частности, такой экспорт реализован для иерархии классов, описывающих рабочие модели системы моделирования процессов образования и распространения загрязнений AirEcology-P. Соответствующая иерархия классов с корневым элементом clsE_Element помимо обычных классов, предусматривающих экспорт, включает экспортирующий класс clsE_Export. Включение в визуальную экологическую модель объекта такого класса приводит к тому, что данная модель порождает, во-первых, заготовки описаний суррогатных классов экологических моделей в одной из подпапок папки Classes, и, во-вторых, в зависимости от значений свойств данного объекта, либо соответствующую XML-модель (включающую только объекты суррогатных классов), либо соответствующий текстовый файл, состоящий из типовых фраз, описывающих суррогатные объекты.

вать, вызывающей стандартный для системы диалог редактирования полей-свойств.

4. Вписать соответствующую фразу в поле **Фраза-Шаблон**. Если предполагается сопоставление не фразе, а признаку конца текста, то в данное поле вписывается одиночный символ **$**.

5. Нажать кнопку с указывающей вниз зеленой стрелкой. При этом полученное сопоставление появится в поле **Сопоставления**.

Для *удаления уже существующего сопоставления* необходимо найти и выделить его в списке поля **Сопоставления**, затем нажать кнопку с указывающей вверх зеленой стрелкой. При этом указанное сопоставление будет удалено из списка **Сопоставления**.

Запуск процесса приобретения знаний

Когда построение списка сопоставлений закончено, можно перейти собственно к процессу приобретения знаний. Для этого необходимо предварительно указать базовые имена файлов CSV-таблиц, которые могут быть сгенерированы в ходе данного процесса, в полях **Базовое имя справочных таблиц** и **Базовое имя трансформирующих таблиц**. Эти базовые имена *должны быть уникальны для каждого проекта*, в противном случае результаты могут быть непредсказуемы.

Далее необходимо нажать кнопку **Сгенерировать распознающие скрипты**, активизирующую процесс приобретения знаний. Данный процесс может быть достаточно длительным (до нескольких минут), его выполнение частично отражается внутренними диагностическими сообщениями в поле **Рабочий журнал**.

В ходе процесса возможно появление сообщений об ошибках, свидетельствующих о невозможности однозначного построения индуцирующих скриптов. В таком случае следует сохранить проект и закрыть окно приобретения знаний.

При успешном завершении процесса приобретения знаний будет выдано сообщение, показанное на рисунке [Рис. П4.2].

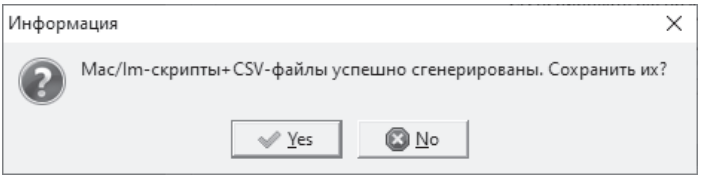


Рис. П4.2. Сообщение об успешном завершении процесса приобретения знаний

При нажатии на кнопку **Yes** система автоматически модифицирует файлы **induct.mac** для классов всех объектов, для которых были определены сопоставления (в файлы будут записаны новые версии скриптов **Auto**), Также будет обнаружен и модифицирован скрипт прямого логического вывода **induct.im** (в него также будет записана новая версия **Auto**). Необходимо обратить особое внимание, что файл **induct.ini** автоматически модифицирован не будет, о чем система выдаст соответствующее предупреждающее сообщение. Тем не менее, предлагаемые изменения для этого файла будут помещены в поле **Предложения в ini-файл**, откуда они могут быть скопированы и помещены в файл в ручном режиме. В заключение будут автоматически созданы или заменены CSV-файлы таблиц, необходимых для поддержки работы сгенерированных индуцирующих скриптов (имена файлов этих таблиц будут включать значения, указанные в полях **Базовое имя …**, и порядковые номера). Данные файлы будут созданы непосредственно в рабочем каталоге системы.

Сгенерированные распознающие скрипты и файл скрипта прямого логического вывода могут быть изменены непосредственно в интерфейсе окна приобретения знаний.

Файл скрипта прямого логического вывода будет содержаться в поле **im-файл**. Поле доступно для редактирования. По окончании редактирования необходимо нажать кнопку **Сохранить**, находящуюся рядом с надписью **im-файл**.

Для редактирования распознающего скрипта какого-либо класса необходимо найти и выделить в поле **Классы** его идентификатор. При этом соответствующий скрипт будет автоматически помещен в поле **Распознающий скрипт** окна. Можно отредактировать его и нажать кнопку **Сохранить**, находящуюся рядом с надписью **Распознающий скрипт**.

Внимание! Система не сохраняет отредактированные скрипты автоматически, поэтому, если произвести изменения в скрипт и сразу же выделить в поле **Классы** другой класс, произведенные изменения не будут сохранены, а в поле редактирования загрузится новый скрипт.

Методика полуавтоматической разработки естественно-языковых интерфейсов

Данная задача не представляет существенной сложности, если каждый объект соответствующей предметной области может быть однозначно отображен в некоторое единственное шаблонное предложение (возможно, имеющее некоторые вариации). Тогда данная задача может быть полностью решена автоматически с помощью описанных выше программных средств. Однако в случае сложных предметных областей (например, моделирования образования и распространения загрязнений), когда одному объекту модели соответствует несколько предложений, полностью автоматического решения нет.

Очевидно, что в таком случае необходимо прибегнуть к помощи неких *суррогатных классов*, каждый из которых может быть описан в точности одним предложением. Тогда можно частично решить задачу приобретения знаний с помощью вышеописанных программных средств, добившись стойкого однозначного распознавания постановки задачи в терминах суррогатных классов. При этом остается вручную написать скрипт обратного логического вывода, который по первичной ОСМ из набора объектов суррогатных классов, однозначно вос-

становит выходную ОСМ в терминах исходных содержательных классов.

Определенную сложность представляет составление обучающих примеров для подсистемы приобретения знаний, которые можно было бы использовать в режиме автоматического сопоставления объектов шаблонным предложениям. Рекомендуем следующую методику:

1. Пишется специальный экспортирующий класс с настройкой «Генерация текста/Генерация XML».

2. Адаптируются порождающие методы обычных классов текущей предметной области – они анализируют структуру текущей модели и, если в ней присутствует объект экспортирующего класса, генерируют не решающую программу, а соответствующее описание текущего объекта в виде XML-фрагмента или нескольких предложений текста на естественном языке (в зависимости от настроек объекта экспортирующего класса). Удобно воспользоваться функцией

ExportXMLElement($n, $ClassID, $ID, $Params)

на языке PHP, которая приведена в конце данного пункта. Функция выдает в выходной поток XML-фрагмент, соответствующий объекту (с номером **$n**) с идентификатором **$ID** класса **$ClassID** и значениями свойств из ассоциативного массива **$Params**. Относительный путь к субиерархии классов-суррогатов должен быть задан в глобальной переменной **$ExportPath**. Следует заметить, что при XML-экспорте порождающая программа также должна самостоятельно генерировать заголовочный тэг, DTD, корневой тэг **<System>** и тэг **<Elements>**.

3. На базе произвольных моделей (с экспортирующими объектами) генерируются обучающие пары «XML - текстовое описание», которые вполне можно использовать в режиме автоматического сопоставления при генерации индуцирующих скриптов.

Что же касается составления скриптов обратного логического вывода, то, обычно, можно ограничиться одним скриптом, который составляется для суррогатного класса с фразой-шаблоном «**$**», обрабатывающего конец входного текста (если такой класс в иерархии отсутствует, то его можно добавить вручную).

Функция XML-экспорта объекта, попутно создающая описание класса в подпапках **Classes**:

```
   function ExportXMLElement($n, $ClassID, $ID, $Params) {
     global $ExportPath;
     if ($ExportPath != "") {
        $Path = "Classes\\" . $ExportPath . "\\" . $ClassID;
        if (!file_exists($Path))
           mkdir($Path);
        $f = fopen($Path . "\\class.ini", "w");
        fwrite($f, "[Definition]\n");
        fwrite($f, "Name=ЭКСПОРТ_".substr($ClassID, 3)."\n");
        fwrite($f, "InheritScript=1\n\n");
        fwrite($f, "[Parameters]\n");
        foreach ($Params as $Key => $Val)
          fwrite($f, "{; Text; '$Key'} $Key=\n");
        fclose($f);
     }
?>
  <Element ClassID="<? echo $ClassID; ?>" ParentID="" ID="<?
echo $ID; ?>" Permanent="False">
   <Show Class="True" Name="True" Image="False"/>
   <Position Left="<? echo ($n % 8)*150; ?>" Top="<? echo
(int)($n/8)*150; ?>"/>
<?
     if (count($Params) == 0) echo "<Parameters NumI-
tems=\"0\"/>\n";
     else {
        echo "<Parameters NumItems=\"".count($Params)."\">\n";
        foreach ($Params as $Key => $Val)
           echo "           <Parameter ID=\"".$Key."\" In-
dent=\"0\">".htmlspecialchars($Val)."</Parameter>\n";
        echo "</Parameters>\n";
     }
?>
   <InternalInputs NumItems="0"/>
   <InternalOutputs NumItems="0"/>
   <PublishedInputs NumItems="0"/>
   <PublishedOutputs NumItems="0"/>
   <InputLinks/>
   <OutputLinks/>
  </Element>
<?
   }
```

Приложение П5. Пример исходной (нераспараллеленной) программы

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>

#pragma auto parallelize
#pragma auto pure(malloc,fabs,free,sizeof,omp_get_wtime)

#define BASE_PARALLEL 1
#define theta 1.83
#define NX 40
#define NY 40
#define h 0.1
#define NP 15000

// Собирающая электростатическая линза
#define U1 200
#define U2 5000

#define e -1.5E-13
#define m 1E-11

#define e0 8.85E-12

#define V (h*h)
#define tau 0.000015
#define T 0.09
#define POISSON_EPS 0.01
#define TOL_EPS 0.25

int main() {
      double * U  = (double *)malloc(NY*NX*sizeof(double));
      double * UU = (double *)malloc(NY*NX*sizeof(double));
      double * EX = (double *)malloc(NY*NX*sizeof(double));
      double * EY = (double *)malloc(NY*NX*sizeof(double));
          double * PX = (double *)malloc(NP*sizeof(double));
          double * PY = (double *)malloc(NP*sizeof(double));
          int * X = (int *)malloc(NP*sizeof(int));
          int * Y = (int *)malloc(NP*sizeof(int));
```

```
	double ro[NY][NX];

	split_private double t;
	split_private double tm;
	split_private int i, j;

	for (i = 0; i < NY; i++)
		for (j = 0; j < NX; j++) {
			UU[i*NX+j] = j == NX-1 ? U2 : j == NX/2 && (i < NY/4 || i >
3*NY/4) ? U1 : 0.0;
			EX[i*NX+j] = 0.0;
			EY[i*NX+j] = 0.0;
		}
	for (i = 0; i < NP; i++) {
		int x, y;

		PX[i] = 0.5*NX*h*rand()/RAND_MAX;
		PY[i] = NY*h*rand()/RAND_MAX;

		x = PX[i]/h;
		y = PY[i]/h;
		if (x < 0) x = 0;
		else if (x > NX-1) x = NX-1;
		if (y < 0) y = 0;
		else if (y > NY-1) y = NY-1;

		X[i] = x;
		Y[i] = y;
	}

	tm = omp_get_wtime();

	for (t = 0.0; t < T; t += tau) {
		unsigned int n[NY][NX] = { 0 };
		double err;
		int ptr = 0;
		for (i = 0; i < NY; i++)
			for (j = 0; j < NX; j++, ptr++)
				U[ptr] = UU[ptr];
		for (i = 1; i < NY - 1; i++)
			for (j = 1; j < NX - 1; j++) {
				EX[i*NX+j] = -(U[i*NX+j+1]-U[i*NX+j-1])/2.0/h;
				EY[i*NX+j] = -(U[(i+1)*NX+j]-U[(i-1)*NX+j])/2.0/h;
			}
```

```
		#pragma omp parallel for if(BASE_PARALLEL) private(i)
num_threads(split_tune)
		for (i = 0; i < NP; i++) {
			PX[i] += tau*e*EX[Y[i]*NX+X[i]]/m;
			PY[i] += tau*e*EY[Y[i]*NX+X[i]]/m;
		}

		#pragma omp parallel for if(BASE_PARALLEL) private(i)
num_threads(split_tune)
		for (i = 0; i < NP; i++) {
			int x = PX[i]/h;
			int y = PY[i]/h;
			if (x < 0) x = 0;
			else if (x > NX-1) x = NX-1;
			if (y < 0) y = 0;
			else if (y > NY-1) y = NY-1;

			Y[i] = y;
			X[i] = x;
			n[y][x]++;
		}

		for (i = 0; i < NY; i++)
			for (j = 0; j < NX; j++)
				ro[i][j] = n[i][j]*e/V;

		do {
			err = 0.0;

				#pragma omp parallel for if(BASE_PARALLEL) private(i,j)
num_threads(split_tune)
				for (i = 1; i < NY - 1; i++)
					for (j = 1+(i-1)%2; j < NX - 1; j+=2) {
					  int ptr = i*NX + j;
					  if (!(j == NX/2 && (i < NY/4 || i > 3*NY/4))) {
						double _new = (1-theta)*UU[ptr] + theta/4.0*(UU[ptr-
1]+UU[ptr+1]+UU[ptr+NX]+UU[ptr-NX]-h*h*ro[i][j]/e0);
						double loc_err = fabs(UU[ptr] - _new);
						if (loc_err > err) err = loc_err;
						UU[ptr] = _new;
					  }
					}
				#pragma omp parallel for if(BASE_PARALLEL) private(i,j)
num_threads(split_tune)
				for (i = 1; i < NY - 1; i++)
					for (j = 1+i%2; j < NX - 1; j+=2) {
```

```
                                int ptr = i*NX + j;
                                if (!(j == NX/2 && (i < NY/4 || i > 3*NY/4))) {
                                        double _new = (1-theta)*UU[ptr] + theta/4.0*(UU[ptr-
1]+UU[ptr+1]+UU[ptr+NX]+UU[ptr-NX]-h*h*ro[i][j]/e0);
                                        double loc_err = fabs(UU[ptr] - _new);
                                        if (loc_err > err) err = loc_err;
                                        UU[ptr] = _new;
                                }
                                }
                        for (j = 0; j < NX; j++) {
                                UU[j] = UU[NX + j];
                                UU[(NY-1)*NX + j] = UU[(NY-2)*NX + j];
                        }
                } while (err > POISSON_EPS);
        }

        for (i = 0; i < NY; i++) {
                for (j = 0; j < NX; j++)
                        printf("%lf\t", UU[i*NX+j]);
                printf("\n");
        }

        return 0;
}
```

Приложение П6. Пример программы, автоматически распараллеленной с применением опосредованной трансформации средствами PGEN++

```
#include "transact.h"
#define split_private /* split-private */
#include <stdlib.h>

#include <stdio.h>
#include <math.h>

#define BASE_PARALLEL 1

#define theta 1.83
#define NX 40
#define NY 40
#define h 0.1
#define NP 15000

#define U1 200
#define U2 5000

#define e -1.5E-13

#define m 1E-11
#define e0 8.85E-12

#define V (h*h)
#define tau 0.000015
#define T 0.09
#define POISSON_EPS 0.01
#define TOL_EPS 0.25

int  main( ){
  double * U  = (double *)malloc(NY*NX*sizeof(double));
  double * UU = (double *)malloc(NY*NX*sizeof(double));
  double * EX = (double *)malloc(NY*NX*sizeof(double));
  double * EY = (double *)malloc(NY*NX*sizeof(double));
  double * PX = (double *)malloc(NP*sizeof(double));
  double * PY = (double *)malloc(NP*sizeof(double));
```

```
  int * X = (int *)malloc(NP*sizeof(int));
  int * Y = (int *)malloc(NP*sizeof(int));
  double ro[NY][NX];
  split_private double t;
  split_private double tm;
  split_private int i, j;
  for ( i = 0; i < NY; i++ )
    for ( j = 0; j < NX; j++ )
      {
        UU[i*NX+j] = j == NX-1 ? U2 : j == NX/2 && (i < NY/4 || i > 3*NY/4) ? U1 : 0.0;
        EX[i*NX+j] = 0.0;
        EY[i*NX+j] = 0.0;
      }
  for ( i = 0; i < NP; i++ )
    {
      int x, y;
      PX[i] = 0.5*NX*h*rand()/RAND_MAX;
      PY[i] = NY*h*rand()/RAND_MAX;
      x = PX[i]/h;
      y = PY[i]/h;
      if ( x < 0 )
        x = 0;
      else
        if ( x > NX-1 )
          x = NX-1;
      if ( y < 0 )
        y = 0;
      else
        if ( y > NY-1 )
          y = NY-1;
      X[i] = x;
      Y[i] = y;
    }
  tm = omp_get_wtime();
#pragma omp parallel num_threads(2) private(t,tm,i,j)
  {
    int __id__ = omp_get_thread_num();
    TOut<double > * out_ro = __id__ == 0 ? new TOut<double >("ro63", (NY)*(NX), 2, 0.01, -
1, "63") : NULL;
    TIn<double > * in_ro = __id__ == 1 ? new TIn<double >("ro63", (NY)*(NX), 2, 0.01, -1,
"63") : NULL;
    for ( t = 0.0; t < T; t += tau )
      {
        unsigned int n[NY][NX] = { 0 };
        double err;
        int ptr = 0;
```

```
      if ( __id__ == 0 )
       {
        for ( i = 0; i < NY; i++ )
          for ( j = 0; j < NX; j++, ptr++ )
            U[ptr] = UU[ptr];
       }
transaction_atomic("63")
      {
       if ( __id__ == 0 )
        {
          for ( i = 1; i < NY - 1; i++ )
            for ( j = 1; j < NX - 1; j++ )
              {
                EX[i*NX+j] = -(U[i*NX+j+1]-U[i*NX+j-1])/2.0/h;
                EY[i*NX+j] = -(U[(i+1)*NX+j]-U[(i-1)*NX+j])/2.0/h;
              }
#pragma omp parallel for if(BASE_PARALLEL) private(i) num_threads(split_tune)

          for ( i = 0; i < NP; i++ )
            {
              PX[i] += tau*e*EX[Y[i]*NX+X[i]]/m;
              PY[i] += tau*e*EY[Y[i]*NX+X[i]]/m;
            }
#pragma omp parallel for if(BASE_PARALLEL) private(i) num_threads(split_tune)

          for ( i = 0; i < NP; i++ )
            {
              int x = PX[i]/h;
              int y = PY[i]/h;
              if ( x < 0 )
                x = 0;
              else
                if ( x > NX-1 )
                  x = NX-1;
              if ( y < 0 )
                y = 0;
              else
                if ( y > NY-1 )
                  y = NY-1;
              Y[i] = y;
              X[i] = x;
              n[y][x]++;
            }
          for ( i = 0; i < NY; i++ )
            for ( j = 0; j < NX; j++ )
              ro[i][j] = n[i][j]*e/V;
```

```
        out_ro->put((double  *)ro);
      }
    else
      {
        double  ro[NY][NX];
        in_ro->get((double  *)ro, 0);
        do
          {
            err = 0.0;
#pragma omp parallel for if(BASE_PARALLEL) private(i,j) num_threads(split_tune)

            for ( i = 1; i < NY - 1; i++ )
              for ( j = 1+(i-1)%2; j < NX - 1; j+=2 )
                {
                  int ptr = i*NX + j;
                  if ( !(j == NX/2 && (i < NY/4 || i > 3*NY/4)) )
                    {
                      double _new = (1-theta)*UU[ptr] + theta/4.0*(UU[ptr-
1]+UU[ptr+1]+UU[ptr+NX]+UU[ptr-NX]-h*h*ro[i][j]/e0);
                      double loc_err = fabs(UU[ptr] - _new);
                      if ( loc_err > err )
                        err = loc_err;
                      UU[ptr] = _new;
                    }
                }
#pragma omp parallel for if(BASE_PARALLEL) private(i,j) num_threads(split_tune)

            for ( i = 1; i < NY - 1; i++ )
              for ( j = 1+i%2; j < NX - 1; j+=2 )
                {
                  int ptr = i*NX + j;
                  if ( !(j == NX/2 && (i < NY/4 || i > 3*NY/4)) )
                    {
                      double _new = (1-theta)*UU[ptr] + theta/4.0*(UU[ptr-
1]+UU[ptr+1]+UU[ptr+NX]+UU[ptr-NX]-h*h*ro[i][j]/e0);
                      double loc_err = fabs(UU[ptr] - _new);
                      if ( loc_err > err )
                        err = loc_err;
                      UU[ptr] = _new;
                    }
                }
            for ( j = 0; j < NX; j++ )
              {
                UU[j] = UU[NX + j];
                UU[(NY-1)*NX + j] = UU[(NY-2)*NX + j];
              }
```

```
            }
            while ( err > POISSON_EPS )
            ;
          }
        }
      }
    delete in_ro;
    delete out_ro;
  }
  for ( i = 0; i < NY; i++ )
    {
      for ( j = 0; j < NX; j++ )
        printf("%lf\t", UU[i*NX+j]);
      printf("\n");
    }
  return 0;
}
```