Российская Академия Наук
Ордена Трудового Красного Знамени
Институт радиотехники и электроники

На правах рукописи

Морозов Алексей Александрович

Логический анализ функциональных диаграмм в процессе интерактивного проектирования информационных систем

05.13.11 — математическое и программное обеспечение вычислительных машин, комплексов, систем и сетей

Диссертация на соискание учёной степени кандидата физико-математических наук

Научный руководитель
доктор
физико-математических
наук **Ю. В. Обухов**

Москва 1998

Оглавление

Введение

В диссертации развит аппарат объектно-ориентированного логического программирования, исследованы и разработаны логические средства для семантического анализа функциональных диаграмм при интерактивном проектировании информационных систем.

Актуальность темы

На современном этапе развития информатики одной из главных проблем является создание программных средств, математически строго выполняющих рутинную работу человека, связанную с разработкой и анализом информационных систем.

Одним из современных методов анализа и проектирования информационных систем является метод описания систем в виде *функциональных диаграмм* (ФД), которые предоставляют разработчику определённые графические средства проектирования. Однако существующие технологии и системы программной поддержки проектирования и анализа ФД обеспечивают лишь проверку синтаксической правильности разрабатываемых диаграмм.

В диссертационной работе впервые решена задача обеспечения *семантической правильности* функциональных диаграмм, а именно:

1. Разработаны средства *семантического анализа* (анализа смысла) диаграмм, учитывающего — в отличие от синтаксического анализа — дополнительную информацию, соответствующую смысловому наполнению анализируемой диаграммы: числовые параметры блоков ФД, правила соединения блоков, зависимость параметров проектируемой системы от параметров блоков и т. п.

2. Предложен и разработан метод выявления и устранения смысловых противоречий в процессе интерактивного построения ФД.

Центральной идеей работы является использование *объектно-ориентированного логического программирования* для семантического анализа графических диаграмм. В диссертации исследованы и разработаны средства объектно-ориентированного логического программирования для семантического анализа ФД, показана перспективность сочетания графических диаграмм, в качестве визуального интерфейса «человек-машина», и средств объектно-ориентированного логического программирования, позволяющих в процессе интерактивного проектирования «оживить» диаграммы с помощью формализованного описания и анализа их семантики.

Цель и задачи работы

Целью диссертационной работы является *исследование и разработка средств логического программирования для анализа ФД в процессе интерактивного проектирования информационных систем.*

Для достижения этой цели поставлены и решены следующие задачи:

- Разработка *логической интерпретации понятий объектно-ориентированного программирования (ООП)*, отражающей его *структурные*, *динамические* и *информационные* аспекты.
- Разработка *метода логического анализа ФД в процессе интерактивного проектирования информационных систем.*
- Разработка *объектно-ориентированного логического языка* для формализованного описания семантики и анализа ФД.

Методы исследования

В работе использовались аппараты и методы математической логики, логического и объектно-ориентированного программирования.

Научные результаты, вынесенные на защиту

В диссертационной работе получены новые научные результаты:

1. Метод логического объектно-ориентированного описания и анализа функциональных диаграмм информационных систем, обеспечивающий семантическую правильность функциональных диаграмм в процессе интерактивного проектирования.

2. Метод повторного доказательства подцелей логической программы, позволяющий интерпретировать в логическом языке интерактивный режим и разрушающее присваивание.

3. Объектно-ориентированный логический язык — Áкторный Пролог.

Научная и практическая ценность

Полученные результаты являются основой для разработки методов визуального объектно-ориентированного логического программирования в области анализа и проектирования информационных систем, функционального и имитационного моделирования, а также быстрого прототипирования и программирования информационных систем.

Предложенные принципы и логические средства являются универсальными и могут быть использованы для анализа других типов диаграмм, в первую очередь, *объектных*.

Созданный прототип реализации Акторного Пролога используется для экспериментов с логическим акторным программированием и может служить основой для разработки промышленной версии языка. Результаты работы внедрены в Институте радиотехники и электроники Российской Академии Наук.

Доклады и печатные публикации

Основные положения работы докладывались на первой и второй международных конференциях по логическому программированию в Иркутске (1990 г.) и Санкт-Петербурге (1991 г.), на 10-й научно-технической конференции «Планирование и автоматизация научных исследований» в Москве (1992 г.), на XI международной конференции «Логика, методология, философия науки» в Обнинске (1995 г.), на II и IV международных конференциях «Развитие и применение открытых систем» в Петрозаводске (1995 г.) и Нижнем Новгороде (1997 г.), а также на международной конференции «Дискретные модели в теории управляющих систем» (Красновидово, 1997 г.).

По материалам диссертации опубликовано семь печатных работ [21]–[27]. Результаты диссертации вошли в отчёты по проекту РФФИ 95-01-00822а, проекту ГКНТ «ПИТ» № 05.05.1191, а также ряда хоздоговорных НИР ИРЭ РАН.

Структура и объём диссертации

Диссертация состоит из введения, четырёх глав, заключения и приложений. Объём работы (за исключением приложений) составляет 134 страницы в формате машинописного текста, в том числе 30 рисунков и 2 таблицы. Список литературы содержит 81 наименование.

Краткое содержание работы

Синтаксис, семантика и проблемы логической интерпретации функциональных диаграмм

В **главе 1** приведён краткий обзор различных видов ФД, обсуждаются семантика ФД и проблемы их логической интерпретации.

Функциональной диаграммой называется графическое изображение, состоящее из блоков и связывающих их дуг, обозначающих, соответственно, некоторые активности (процессы) и потоки объектов (данных), передаваемых между этими активностями. Исторически ФД возникли в качестве средства описания систем (в том числе информационных) и в настоящее время широко применяются в таких областях как:

- структурный системный анализ и проектирование (функциональное моделирование) — SA-диаграммы в SADT [19] и IDEF0, диаграммы потоков данных [11] Гейна-Сарсона и др.;

- визуальное программирование измерительных информационных систем — диаграммы LabView [30, 60] и др.

Современные тенденции развития информатики, такие как *открытые информационные системы* [6, 32], *объектно-ориентированный подход к анализу, проектированию и программированию, повторное использование компонентов* [43, 4, 68, 69] создают предпосылки для разработки новых методов и средств анализа и проектирования, обеспечивающих *семантическую правильность* проектируемых диаграмм.

Одним из наиболее перспективных подходов к разработке и реализации таких методов и средств является сочетание графических диаграмм для поддержки визуального интерфейса «человек-машина» и средств объектно-ориентированного логического программирования, позволяющих «оживить» диаграммы с помощью формализованного описания и

анализа их семантики в процессе интерактивного проектирования. Однако анализ существующих логических подходов и средств [39, 2, 16, 14, 36, 64, 69, 3] показывает необходимость решения следующих проблем теоретического характера, затрудняющих использование логики для поддержки интерактивного проектирования информационных систем:

- Отсутствие вполне развитого и общепринятого подхода к интерпретации в логических языках объектов и связей между ними.
- Недостаточная теоретическая проработка вопросов взаимодействия человека и компьютера в логическом программировании.
- Проблема поддержки согласованного состояния объектов в моделях информационных систем.

Метод интерактивного функционального моделирования информационных систем и Акторный Пролог

В **главе 2** диссертационной работы предложен *метод интерактивного функционального моделирования*, который включает следующий план действий:

1. Одновременно с разработкой ФД информационной системы осуществляется построение программы на специальном логическом языке, описывающей блоки и связи ФД в терминах логического программирования. В диссертации показано, что следует выделить, по крайней мере, три разновидности таких логических программ — мы будем называть их *семантическими моделями*:
   (a) семантические модели структуры ФД информационной системы — *структурные семантические модели*, предназначенные для проверки и логического доказательства правильности соединения компонентов, а также анализа свойств системы;
   (b) *интерактивные семантические модели*, обеспечивающие семантическую правильность ФД проектируемой системы в процессе её интерактивного редактирования;
   (c) *имитационные семантические модели* для моделирования работы системы, которые могут также рассматриваться в качестве парадигмы объектно-ориентированного логического программирования информационных систем.

Построение семантической модели может происходить частично автоматически, с помощью трансляции разрабатываемой ФД, а частично — «вручную», путём формализации знаний экспертов о семантике составных частей диаграммы.

2. Начиная с некоторого момента, когда накоплено достаточно информации о блоках и связях, а также о правилах соединения блоков ФД (то есть построение семантической модели, хотя бы в первом приближении, завершено), построенная логическая программа может быть исполнена. В дальнейшем может быть осуществлён один из двух вариантов использования этой программы:

   (a) Программа исполняется время от времени для анализа заданных параметров информационной системы или проверки правильности соединения функциональных блоков. Такой режим использования лучше подходит, скорее, для структурных и имитационных семантических моделей.

   (b) Программа реализует интерактивный режим работы проектировщика с машиной, реагируя на изменения, вносимые им в структуру ФД или параметры её составных частей, и проверяя правильность соединения и согласованность режимов работы компонентов моделируемой системы.

3. Семантическая модель является «отторгаемым продуктом» и может быть передана другому проектировщику, при этом логические правила соединения блоков могут быть сокрыты в описании компонентов ФД, что позволяет использовать их как «чёрные ящики». Более того, могут быть отдельно переданы компоненты семантической модели, разработанные «вручную», что позволит другому проектировщику автоматически (с помощью трансляции своих ФД) строить свои собственные семантические модели.

Для построения и исполнения различных видов семантических моделей в диссертации разработан Акторный Пролог [26, 22] — объектно-ориентированный логический язык, реализующий обобщённую *логическую акторную модель ФД информационной системы, проектируемой в интерактивном режиме.*

В качестве обобщённой модели *ФД информационной системы, проектируемой в интерактивном режиме*, разработана *логическая акторная модель* [23] (см. рис. 1), представляющая диаграмму в виде некоторой

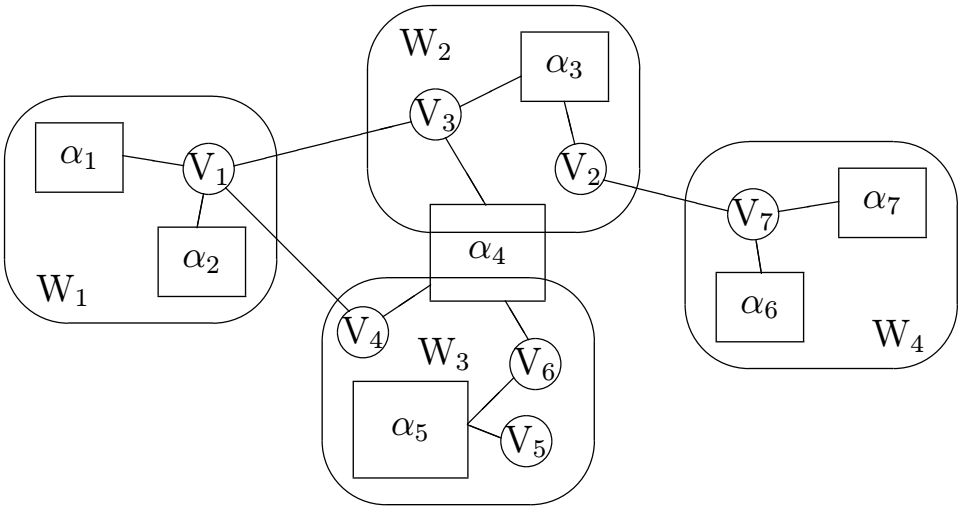


Рис. 1: Логическая акторная модель ФД.

теоремы на логическом языке, разделённой на *логические акторы — повторно доказываемые* подцели ($\alpha_1, \ldots, \alpha_n$ на рисунке), взаимодействующие через *общие переменные* ($\mathrm{V}_1, \ldots, \mathrm{V}_m$). Доказательство этой теоремы осуществляется в некотором *объектно-ориентированном пространстве поиска*, состоящем из отдельных миров ($\mathrm{W}_1, \ldots, \mathrm{W}_k$), топология которого соответствует структуре ФД моделируемой системы.

С точки зрения логики, взаимодействие человека с компьютером в процессе интерактивного построения и анализа ФД удобно рассматривать как доказательство некоторой теоремы о свойствах проектируемой системы, в котором одновременно принимают участие человек, изменяющий условия задачи, и машина, обеспечивающая корректность доказательства, приводя его по мере необходимости в соответствие с изменяемыми исходными данными.

Логическая акторная модель вычислений, соответствующая такому представлению ФД информационной системы, основана на *повторных доказательствах акторов*, осуществляемых при изменении значений общих переменных, в целях обеспечения *корректности и полноты доказательства теоремы*.

Взаимодействие между акторами реализуется с помощью операции «корректного разрушающего присваивания», изменяющей значения общих переменных программы. В ходе этой операции (например, в процессе доказательства специального предиката корректного разрушающего присваивания *':='*) осуществляется необходимое изменение значений общих переменных, а затем вызывается повторное доказательство акторов,

зависящих от старых значений общих переменных. Если повторное доказательство всех этих акторов завершается успехом, исполнение операции также заканчивается успехом (в этом случае предикат разрушающего присваивания считается истинным, и его доказательство завершается успехом), в противном случае в логической программе происходит откат.

Стратегия управления, обеспечивающая корректность доказательства теоремы при изменении значений общих переменных и произвольном порядке доказательства акторов, названа *механизмом повторного доказательства* (*акторным механизмом*).

В Акторном Прологе *структурные*, *динамические* и *информационные* аспекты ООП реализованы с помощью:

- классов, миров и наследования;
- акторного механизма;
- простых и составных термов.

Точно так же как чистый Пролог является всего лишь иной формой записи некоторых формул логики предикатов первого порядка, все новые синтаксические конструкции, введённые в настоящей работе, являются видоизменением чистого Пролога и, в этом смысле, также эквивалентны формулам логики предикатов. Таким образом, понятия «актор», «класс», «недоопределённое множество» и т. п., используемые в данной работе, являются — так же как и многие другие понятия в логическом программировании — не более чем логическими аналогами соответствующих понятий процедурного программирования.

Использование названных средств языка в процессе функционального моделирования основано на следующих принципах [24]:

1. Каждый функциональный блок диаграммы описывается *экземпляром некоторого класса* логического языка, содержимым которого являются правила, описывающие семантику блока.
2. Связи между блоками функциональной диаграммы описываются с помощью *слотов* экземпляров классов. Для этих же целей удобно использовать *правила второго порядка* [26], имитируемые в Акторном Прологе с помощью *недоопределённых множеств* [26, 22].
3. *Логические акторы* и *механизм повторного доказательства* служат для поддержки интерактивного проектирования функциональных диаграмм.

Модель системы, собранная из отдельных классов, обладает как *операционной*, так и *декларативной семантикой* и может быть однозначно выражена в терминах логики предикатов первого порядка, что позволяет обеспечивать *логическую корректность* проводимого анализа диаграмм. Вторым важным свойством семантических моделей является возможность обеспечивать *логическую полноту* проводимого анализа. Другими словами, если моделируемая система обладает некоторым (возможно, неблагоприятным) свойством, обнаружение этого свойства гарантируется, хотя для этого, возможно, придётся проанализировать большое количество режимов её работы.

Основные усилия в процессе разработки Акторного Пролога были направлены на получение по возможности более чёткого определения синтаксиса и семантики, допускающего простую и эффективную реализацию. Акторный Пролог отвечает требованиям, предъявляемым к современным языкам программирования, в том числе, требованиям к средствам раздельной трансляции и обработки исключительных ситуаций. Одновременно с разработкой языка были созданы прототипы его интерпретатора, браузера (оболочки для визуального программирования), а также транслятора в исходный текст на языке PDC PROLOG.

В приложении к диссертации приведено полное определение Акторного Пролога [22] от 22 марта 1998 г.

Декларативная и операционная семантика Акторного Пролога

В **главе 3** рассмотрен метод повторного доказательства подцелей, определены декларативная и операционная семантики Акторного Пролога. Доказаны теоремы, гарантирующие корректность логической интерпретации ООП в Акторном Прологе.

В основе Акторного Пролога лежит *метод повторного доказательства подцелей*, идея которого заключается в следующем:

1. В логический язык вводятся специальные обозначения, с помощью которых в программе выделяются некоторые подцели (*логические акторы*), связанные общими переменными.

2. Исполнение логических программ осуществляется под управлением специальной стратегии (называемой в Акторном Прологе *механизмом повторного доказательства* или *акторным механизмом*), предусматривающей:

(a) Возможность *изменения значений некоторых общих переменных* и последующего *повторного доказательства зависящих от них акторов* (если это понадобится для восстановления корректности и полноты логического вывода).

(b) Возможность *параллельного исполнения повторных доказательств акторов*, когда это не нарушает корректность и полноту логического вывода.

(c) Возможность *задержки восстановления состояний некоторых акторов при откате программы*, когда это не нарушает корректность и полноту логического вывода (эта идея является основой логической интерпретации *необратимых процессов* в Акторном Прологе).

Параллельное исполнение и задержка восстановления акторов при откате являются факультативными возможностями, поэтому, в целях упрощения изложения, формальное определение операционной семантики Акторного Пролога в диссертации построено на основе некоторой *последовательной абстрактной машины логического вывода*.

Входным языком разработанной абстрактной машины логического вывода является хорновское подмножество формул логики предикатов первого порядка, обогащённого синтаксическими средствами для реализации акторов. Все остальные объектно-ориентированные средства Акторного Пролога, а именно, средства, отражающие структурный и информационный аспекты ООП, однозначно и эффективно реализованы с помощью заданного входного языка.

Логическая корректность такой реализации обосновывается доказанными в работе теоремами:

Теорема 3.1.1. Программа на Акторном Прологе без средств управления, в которой используются «недоопределённые множества», может быть эффективно преобразована (с сохранением операционной семантики) в программу, не содержащую «недоопределённые множества».

Теорема 3.1.2. Программа на Акторном Прологе, в которой используются «предикаты с переменным числом аргументов», может быть эффективно преобразована (с сохранением операционной семантики) в программу, не содержащую «предикаты с переменным числом аргументов».

Теорема 3.1.3. Вещественные числовые литералы в программе на Акторном Прологе могут быть (эффективно) представлены в виде термов логики предикатов первого порядка.

Теорема 3.1.4. Программа на Акторном Прологе, в которой используются «объявления функций» и «вызовы функций», может быть эффективно преобразована (с сохранением операционной семантики) в программу, не содержащую «объявления и вызовы функций».

Теорема 3.1.5. Программа на Акторном Прологе без средств управления и акторных префиксов может быть эффективно преобразована (с сохранением операционной семантики) в программу на чистом Прологе (в формулу хорновского подмножества логики предикатов первого порядка).

Примечание. Средствами управления Акторного Пролога являются отсечение '!' и другие встроенные операторы, заголовки внешних предложений, а также специальный акторный префикс «&».

Определение 3.1.6. *Декларативной семантикой* программы на Акторном Прологе без средств управления является декларативная семантика программы на чистом Прологе, соответствующей по теореме 3.1.5 исходной программе без акторных префиксов.

Теорема 3.1.7 (о корректности). Механизм повторного доказательства Акторного Пролога с проверкой вхождений, без средств управления является корректной стратегией управления.

Теорема о корректности акторного механизма обеспечивает возможность построения *логической интерпретации интерфейса «человек-машина»*, а также разработки соответствующих *средств поддержки интерактивного режима*, гарантирующих корректность результатов, получаемых в ходе совместной деятельности человека и машины.

Теорема 3.1.9 (о полноте). Если P_A — программа на Акторном Прологе без средств управления, а P_S — программа на чистом Прологе, соответствующая по теореме 3.1.5 программе P_A без акторных префиксов, то для любого успешного вычисления C_S в полном дереве поиска программы P_S, исполняемой под управлением стандартной стратегии, в полном дереве поиска программы P_A найдётся некоторое успешное вычисление C_A, выдающее тот же ответ, что и вычисление C_S.

3.1.10. Следствие теоремы о полноте. Программа на Акторном Прологе без средств управления путём перебора вариантов обязательно найдёт *все* существующие решения задачи, если в ходе её исполнения не возникнут бесконечные вычисления.

Заметим, что механизм повторного доказательства *не является справедливой стратегией управления*, поскольку в Акторном Прологе используется поиск в глубину, как в «обычном» Прологе, и программы на Прологе могут зацикливаться.

Акторный механизм позволяет интерпретировать в логическом языке такие понятия как разрушающее присваивание и необратимый процесс и, таким образом, является решением *проблемы фрейма* [17, 27], свободным от недостатков немонотонных логических систем [33].

Необходимо также отметить, что идея локализации значений общих переменных в акторах, предложенная и реализованная в Акторном Прологе, позволяет организовать логический вывод в *распределённых системах*, в условиях противоречивости и несвоевременного обновления информации. Таким образом, разработанный логический язык является мощным средством *поддержки истинности* [80], обеспечивающим декларативную семантику системы акторов, инвариантную по отношению к возможному недетерминизму её поведения [21] — он является инструментом для логического программирования *открытых систем искусственного интеллекта* [37, 65].

Использование Акторного Пролога для анализа ФД информационных систем

В **главе 4** обсуждается один из возможных подходов к реализации интерактивного функционального моделирования средствами Акторного Пролога, рассматривается пример простейшей интерактивной семантической модели.

Поддержка в Акторном Прологе средств ООП позволяет использовать преимущества ООП при проектировании и семантическом анализе функциональных диаграмм, а именно:

- Позволяет локализовать и «спрятать» составные части семантической модели в отдельные блоки, так чтобы затем эти блоки можно было использовать в процессе разработки функциональной диаграммы в качестве «чёрных ящиков», не вникая в их логическую «начинку».

- Позволяет использовать при разработке семантических моделей функциональных диаграмм принципы *абстракции* и *наследования* (а также допускает *повторное использование описаний* функциональных блоков — в других диаграммах).

- Позволяет в значительной степени автоматизировать разработку семантических моделей с помощью *автоматической трансляции графических функциональных диаграмм* в текст на объектно-ориентированном логическом языке.

- Позволяет реализовать *интерактивный режим работы* с проектировщиком или читателем функциональной диаграммы.

Благодарности

Я хотел бы поблагодарить за помощь и поддержку моего научного руководителя Ю. В. Обухова, а также А. Ф. Полупанова, А. Я. Олейникова, В. В. Алексеева, А. Н. Комарова, С. А. Хлебникова, А. С. Венецкого, В. А. Захарова, И. В. Горскую и М. В. Захарьящева, принимавших участие в обсуждении работы, В. А. Петухина, А. В. Манциводу, А. Г. Петропавловского и А. В. Бударова, оказавших влияние на начальном этапе проекта.

Особую благодарность я хотел бы высказать Н. Л. Новожилову, предложившему идею логического анализа SADT-диаграмм ещё в 1986 году.

Работа выполнена при поддержке Российского Фонда Фундаментальных Исследований (проект 95-01-00822а).

Глава 1

Синтаксис, семантика и проблемы логической интерпретации функциональных диаграмм

Функциональной диаграммой называется графическое изображение, состоящее из блоков и связывающих их дуг, обозначающих, соответственно, некоторые активности (процессы) и потоки объектов (данных), передаваемых между этими активностями.

Исторически функциональные диаграммы возникли в качестве средства описания систем (в том числе информационных) и применялись в рамках различных методов структурного анализа (анализа требований) и проектирования систем, наиболее известными из которых являются SADT и его подмножество IDEF0 [19], структурный анализ и проектирование Йодана/Де Марко, структурный системный анализ Гейна-Сарсона и др. [11].

Первые применения функциональных диаграмм были чисто «бумажными», так как возможность компьютерной реализации работы с графическими диаграммами появилась лишь в конце 70-х годов — значительно позже разработки первых методов структурного анализа и проектирования. Однако в настоящее время созданы многочисленные компьютерные реализации таких методов, функциональные диаграммы используются в новых методах анализа и проектирования, специально созданных для разработки программ и информационных систем

(таких как ООА — объектно-ориентированный анализ [43]), в составе CASE-средств [11, 54, 9, 3, 69] и средств визуального программирования [41, 70, 76, 30, 60].

Во всех перечисленных областях применения функциональные диаграммы служат, как правило, в качестве:

- средств визуализации идей разработчика;
- средств общения между разработчиками, а также между разработчиками и заказчиками систем;
- средств документирования принятых проектных решений.

Ещё одной ролью функциональных диаграмм, появившейся в связи с использованием вычислительных машин, является

- роль средства общения между разработчиком (проектировщиком системы или просто программистом) и компьютером.

В настоящей работе, посвящённой проблемам интерактивного проектирования, функциональные диаграммы будут рассматриваться именно с точки зрения этой их роли — роли языка общения человека и машины. Поэтому, рассматривая разновидности функциональных диаграмм и методы их анализа, мы будем намеренно абстрагироваться от структурных методов, в рамках которых были разработаны те или иные системы обозначений (причём, как правило, разработаны независимо от вопросов компьютерной реализации).

Разумеется, не следует думать, что предлагаемые в настоящей работе принципы анализа функциональных диаграмм могут быть механически включены в существующие методы структурного анализа и проектирования. Речь идёт, скорее, о разработке новых методов, отражающих и поддерживающих современные тенденции развития информатики и необходимых для расширения области применения структурного анализа и проектирования в соответствии с современными подходами к разработке и использованию информационных систем.

Обсуждение этих вопросов будет продолжено в разделе «Семантика функциональных диаграмм», а теперь мы рассмотрим некоторые наиболее известные разновидности функциональных диаграмм.

1.1 Синтаксис диаграмм

Основными признаками, по которым можно классифицировать различные функциональные диаграммы, являются:

- разновидности блоков, используемых в составе диаграммы;
- разновидности дуг, соединяющих блоки.

Кроме того, функциональные диаграммы можно классифицировать по областям их применения:

- структурный системный анализ и проектирование (функциональное моделирование) — SA-диаграммы в SADT и IDEF0, диаграммы потоков данных Гейна-Сарсона и др.;
- визуальное программирование измерительных информационных систем — диаграммы LabView и др.

Рассмотрим примеры функциональных диаграмм, относящихся к названным областям применения.

1.1.1 Функциональные диаграммы Росса

Одной из самых первых и наиболее известных разновидностей функциональных диаграмм можно назвать SA-диаграммы [19], разработанные в 50-х годах Дугласом Россом в Массачусетском технологическом институте (МТИ). В SA-диаграммах используется всего один тип блоков (так называемые SA-блоки), при этом, однако, «устройство» самого SA-блока и принцип его использования можно назвать открытием в области структурного анализа и проектирования.

SA-блок изображается в виде прямоугольника (рис. 1.1) и сопровождается текстом на естественном языке, описывающим соответствующую активность (названиями блоков служат глаголы и глагольные обороты). Каждая сторона SA-блока имеет вполне определённое особое значение: левая сторона блока предназначена для *входов*, верхняя — для *управления*, правая — для *выходов*, нижняя — для *механизмов* (*исполнителей*) функции.

Такое обозначение отражает следующий принцип активности:

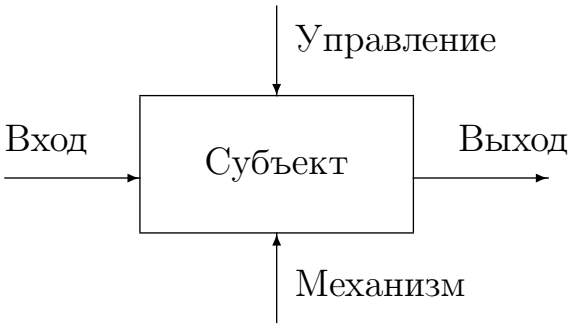


Рис. 1.1: SA-блок.

> входы преобразуются в выходы, управления ограничивают или предписывают условия выполнения, механизмы описывают, за счёт чего выполняются преобразования.

Блоки на диаграмме размещаются по «ступенчатой» схеме в соответствии с их *доминированием*, которое понимается как влияние, оказываемое одним блоком на другие. Кроме того, блоки должны быть пронумерованы, например, в соответствии с их доминированием. Номера блоков служат однозначными идентификаторами для активностей.

Дуги в SA-диаграммах представляют наборы предметов или передаваемые данные и описываются (помечаются) существительными или существительными с определениями. Предметы (данные) могут состоять с активностями в четырёх возможных отношениях: вход, выход, управление, механизм. Каждое из этих отношений изображается дугой, связанной с определённой стороной блока — таким образом стороны блока чисто графически классифицируют предметы, изображаемые дугами. Входные дуги изображают предметы (данные), используемые и преобразуемые активностями. Управляющие дуги обычно изображают информацию, управляющую действиями активностей. Выходные дуги изображают предметы (данные), в которые преобразуются входы. Механизмы обычно изображают физические аспекты функций.

Взаимовлияние блоков может выражаться либо в пересылке *выхода* к другой активности для дальнейшего преобразования, либо в выработке управляющей информации, предписывающей, что именно должна делать другая активность. В SA-диаграммах требуется только пять типов взаимосвязей между блоками для описания их отношений: *управление*, *вход*, *обратная связь по управлению*, *обратная связь по потоку данных*, *выход–механизм*.

Отношения *управления* и *входа* являются простейшими, поскольку они отражают интуитивно очевидные прямые воздействия. Отношение *управления* возникает тогда, когда *выход* некоторого блока непосредственно влияет на *управление* блока с меньшим доминированием. Отношение *входа* возникает тогда, когда *выход* блока становится *входом* для блока с меньшим доминированием.

Обратные связи более сложны, так как они отражают итерацию или рекурсию — *выходы* из одной активности влияют на будущее выполнение других функций, что впоследствии влияет на исходную активность. *Обратная связь по управлению* возникает, когда *выход* некоторого блока влияет на *управление* блока с бóльшим доминированием, а отношение *обратной связи по данным* имеет место, когда *выход* блока становится *входом* другого блока с бóльшим доминированием.

Отношение *выход–механизм* отражает ситуацию, при которой *выход* одной активности становится средством достижения цели (*механизмом*) другой активности.

Дуги в SA-диаграммах, как правило, изображают наборы предметов (данные), поэтому они могут разветвляться и соединяться вместе различным образом. Разветвления дуги означают, что часть её содержимого (или весь набор предметов) может появиться в каждом ответвлении дуги. Дуга всегда помечается до разветвления, чтобы дать название всему набору. Кроме того, каждая ветвь дуги может быть помечена в соответствии со следующими правилами: считается, что непомеченная ветвь содержит все предметы, указанные в метке перед разветвлением; каждая метка ветви уточняет, что именно содержит эта ветвь. Слияние дуг указывает, что содержимое каждой ветви участвует в формировании содержимого объединённой дуги. После слияния дуга всегда помечается для указания нового набора, кроме того, каждая ветвь перед слиянием также может быть помечена: считается, что непомеченные ветви содержат все предметы, указанные в общей метке после слияния; каждая метка ветви уточняет, что именно содержит эта ветвь.

SA-диаграмма является основным рабочим элементом SADT (Structured Analysis and Design Technique) — метода[1] структурного анализа и проектирования систем [19, 11], разработанного в 1969–1973 гг. фирмой SofTech Inc. Второй важнейшей составной частью SADT является идея *иерархической декомпозиции сверху вниз*. В SADT модель систе-

[1]В литературе по структурному анализу термин technique часто переводится также словом «методология».

мы объединяет и организует отдельные SA-диаграммы в иерархические древовидные структуры, при этом требуется, чтобы в каждой отдельной диаграмме было 3–6 блоков: в этих пределах диаграммы удобны для чтения, понимания и использования.

Пример SA-диаграммы, изображающей процесс создания и рецензирования SADT-модели, приведён на рис. 1.1.1.

Корректность SADT-модели проверяется в процессе итеративного рецензирования, когда автор и эксперт многократно совещаются (устно и письменно) относительно достоверности создаваемой модели. На практике, над различными частями модели могут совместно работать множество авторов, потому что каждый функциональный блок модели представляет отдельный субъект, который может быть независимо проанализирован и декомпозирован. Таким образом, модель сама координирует работу коллектива авторов, в то время как процесс SADT-моделирования координирует совместное рецензирование возникающих идей.

SADT успешно использовалась в таких областях как программное обеспечение телефонных сетей, системная поддержка и диагностика, долгосрочное и стратегическое планирование, автоматическое производство и проектирование, конфигурация компьютерных систем, обучение персонала, встроенное программное обеспечение для оборонных систем, управление финансами, материально-техническое снабжение и др. В программе интегрированной компьютеризации производства (ICAM) Министерства обороны США была признана полезность SADT, что привело к стандартизации его части, называемой IDEF0 (в отличие от IDEF0, SADT вместе с функциональными диаграммами использует также модели данных).

1.1.2 Диаграммы потоков данных

В нотациях Йодана и Гейна-Сарсона основным средством описания функциональных требований являются *диаграммы потоков данных* (ДПД) [11, 54].

На ДПД функциональные требования представляются с помощью «процессов» и «хранилищ», связанных потоками данных. Основные символы ДПД изображены на рис. 1.2.

Потоки данных являются механизмами, использующимися для моделирования передачи информации (или физических компонентов) из одной части системы в другую. Потоки на диаграммах обычно изображаются именованными стрелками, ориентация которых указывает направление

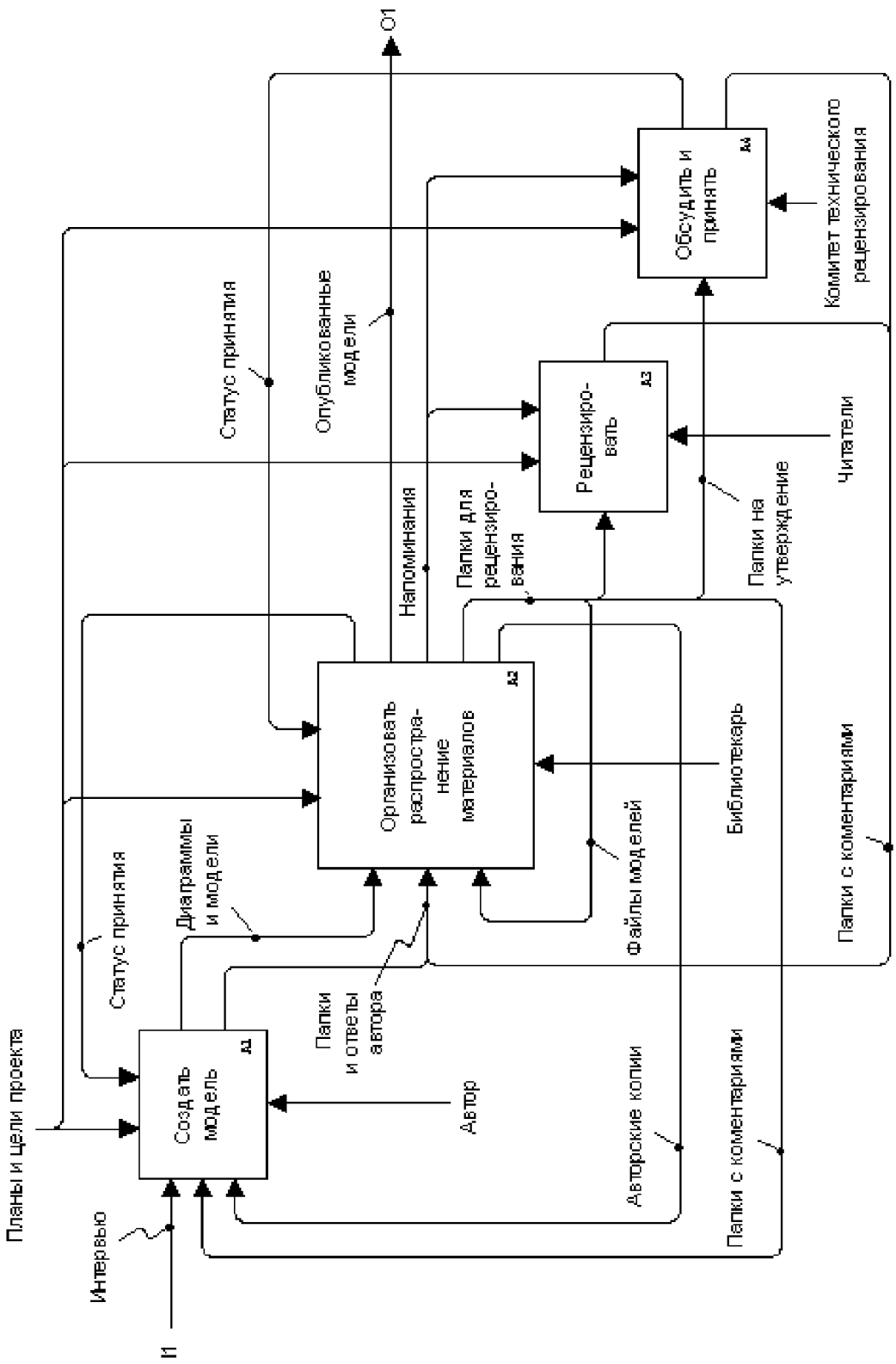


Рис. 1.2: Пример SA-диаграммы.

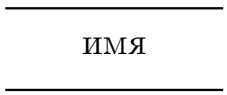
символы	**нотация Йодана**	**нотация Гейна-Сарсона**
поток данных		
процесс		
хранилище		
внешняя сущность		

Рис. 1.2: Основные символы диаграмм потоков данных.

движения информации. Иногда информация может двигаться в одном направлении, обрабатываться и возвращаться назад в её источник. Такая ситуация может моделироваться либо двумя различными потоками, либо одним — двунаправленным.

Назначение *процесса* состоит в продуцировании выходных потоков из входных в соответствии с действием, задаваемым именем процесса. Это имя должно содержать глагол в неопределённой форме с последующим дополнением (например, «вычислить максимальное напряжение»). Кроме того, каждый процесс должен иметь уникальный номер для ссылок на него внутри диаграммы.

Хранилище (накопитель) данных позволяет определять данные, которые будут сохраняться в памяти между процессами. Информация, которую оно содержит, может использоваться в любое время после её определения, при этом данные могут выбираться в любом порядке. Имя хранилища — некоторое существительное — должно идентифицировать его содержимое. В случае когда поток данных входит/выходит в/из хранилища, и его структура соответствует структуре хранилища, он должен иметь то же самое имя, которое нет необходимости отражать на диаграмме.

Внешняя сущность (терминатор) представляет сущность вне контекста системы, являющуюся источником или приёмником системных данных. Её имя должно содержать существительное, например, «источник сигналов». Предполагается, что объекты, представленные такими узлами, не должны участвовать ни в какой обработке.

Декомпозиция ДПД осуществляется на основе процессов: каждый процесс может раскрываться с помощью диаграммы нижнего уровня. Процесс декомпозиции продолжается до тех пор, пока не появится возможность описывать процессы с помощью коротких (до одной страницы) спецификаций (*миниспецификаций обработки*).

Для дополнения ДПД средствами описания управляющих аспектов в системах реального времени, используются дополнительные символы — *управляющий поток*, *управляющий процесс*, *управляющее хранилище* [11]. Графические обозначения названных символов соответствуют обозначениям, приведённым на рис. 1.2, однако вместо непрерывных линий используются пунктирные.

Для представления одного и того же фрагмента данных потоками разных типов используется *узел изменения типа* (рис. 1.3).

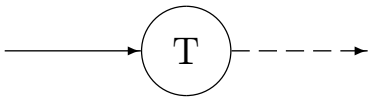


Рис. 1.3: Узел изменения типа.

1.1.3 Функциональные диаграммы систем виртуальных приборов

В отличие от функциональных диаграмм, применяемых в структурных методах анализа и проектирования, функциональные диаграммы, разработанные фирмой National Instruments для программного пакета LabView [30, 60], предназначены для создания, отладки и исполнения прикладных программ. Программы, создаваемые в LabView, разработчики называют *виртуальными приборами*. Каждый такой прибор имеет окно — так называемую *переднюю панель*. Передняя панель служит для управления виртуальным прибором и ввода информации (на рис. 1.4 приведён пример передней панели виртуального прибора). Собственно программа, находящаяся во втором окне, получает от передней панели управляющую информацию и передаёт ей результаты работы.

Программы — то есть функциональные диаграммы — в LabView строятся из достаточно обширного набора «кирпичиков», многие из которых определены заранее. Это, в частности, переменные и константы разнообразных типов и форматов, операции над ними — арифметические, сравнения, строковые, обмена с файлами и др. Существуют такие элементы как датчики случайных чисел, драйверы для работы с лабораторным оборудованием, различные фильтры и преобразования. В качестве элементов могут выступать и ранее созданные программистом виртуальные приборы.

Каждому элементу соответствует своё графическое обозначение (*пиктограмма*). Пиктограммы очень выразительны и легко запоминаются: например, все пиктограммы, относящиеся к беззнаковым целым числам различных форматов, имеют один и тот же синий цвет и характерную рамку, а внутри рамки — указание на формат. Разработка программы сводится к тому, чтобы выбирать из меню нужные элементы, размещать их в удобной последовательности и соединять линиями — «проволокой». Если элементы имеют несовместимые типы, соединить их не удастся. Линии, которые уже подсоединены, имеют различный *цвет* и *толщину*, в зависимости от типов данных, которые по ним передаются. Если при передаче значения возникает преобразование данных разных типов, то в

месте соединения проволоки с элементом появится указывающее на такое преобразование *утолщение*.

Привычные структуры *цикла*, *условного оператора* и *оператора выбора* имеют вид прямоугольников, размеры которых можно произвольно менять. Если какое-либо действие должно выполняться в цикле, его необходимо просто поместить внутрь цикла. Если используется while-цикл, то условие его окончания помещается внутрь цикла и соединяется линией с квадратиком, обозначающим проверку этого условия. Если в цикле происходит, например, считывание данных с какого-либо прибора через определённые промежутки времени, внутрь цикла достаточно поместить пиктограмму, изображающую *метроном*, и написать рядом время задержки (рис. 1.5).

Условный оператор и оператор выбора очень похожи: это прямоугольники с указанием текущего значения в верхней части рамки. При изменении этого значения, внутри рамки становятся видны действия, соответствующие данному значению. Для того чтобы задать вариант, который должен выполняться, к выделенному месту на рамке подводится линия от проверки условия, управляющего рассматриваемым выражением.

В LabView не существует никаких ограничений на создание в рамках одного виртуального прибора независимых друг от друга фрагментов программы и их одновременное выполнение. Это означает, что любой виртуальный прибор может функционировать в многозадачном режиме, поэтому в функциональных диаграммах LabView предусмотрены средства синхронизации. Любой фрагмент программы начинает выполняться только в тот момент, когда все данные для него готовы. Кроме того, для синхронизации можно использовать специальный элемент — прямоугольник, похожий на кусочек кинопленки.

На рис. 1.5 приведён пример функциональной диаграммы, описывающей процесс генерации и приёма сигналов.

Для проверки синтаксической правильности построенной диаграммы в пакете LabView предусмотрен режим проверки синтаксиса. Чаще всего встречаются ошибки двух типов — «есть несоединённые элементы» и «есть плохие линии».

1.2 Семантика диаграмм

Удобство использования и выразительность функциональных диаграмм в большой степени являются следствием того, что они позволяют пред-

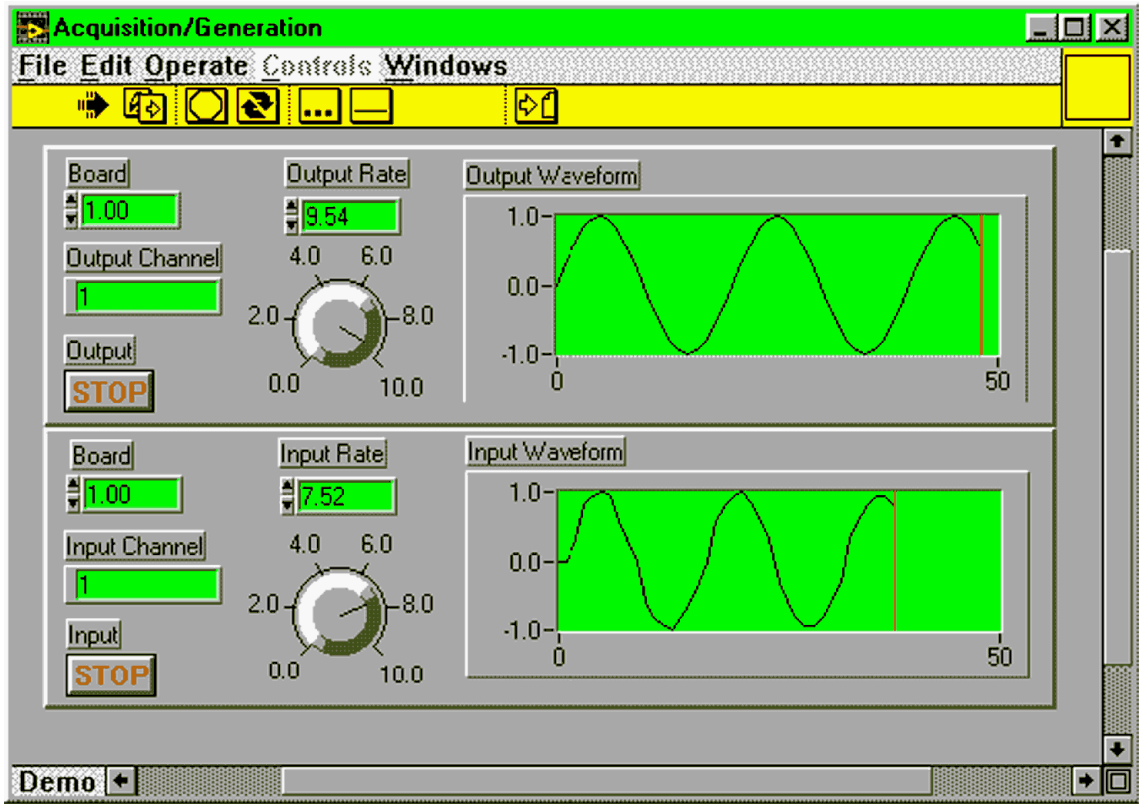


Рис. 1.4: Пример передней панели виртуального прибора.

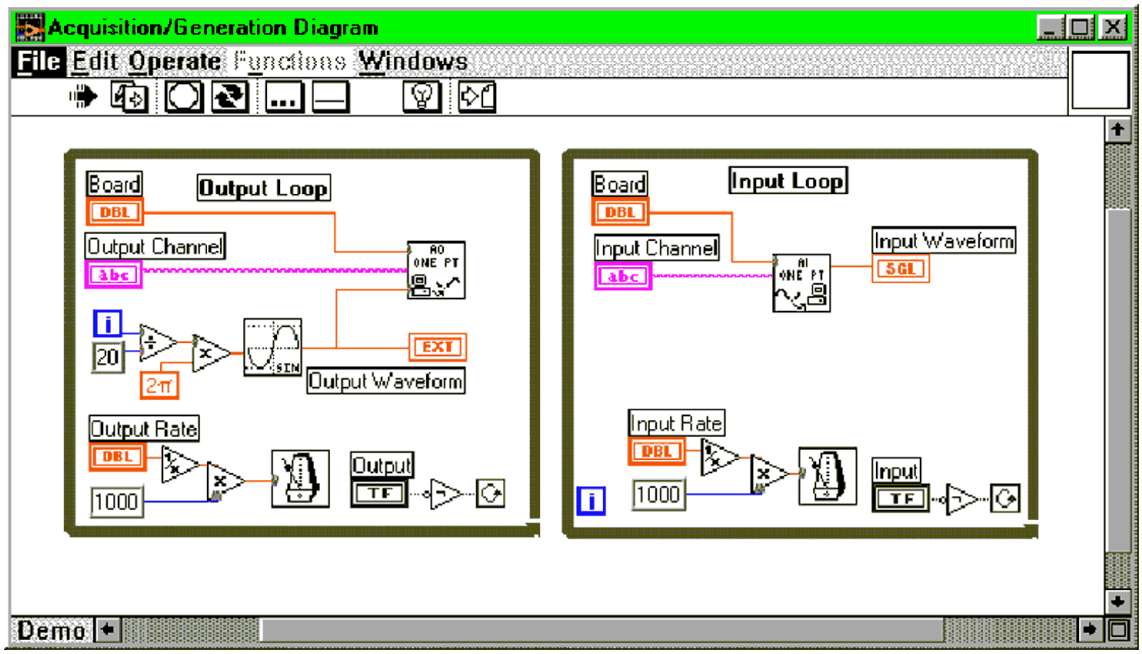


Рис. 1.5: Пример диаграммы виртуального прибора.

ставлять богатое смысловое содержание в компактной графической форме, то есть с помощью графического синтаксиса. Так, например, разработчики метода SADT декларируют возможность полного и точного описания всех статических аспектов[2] функционального содержания системы [19]. Диаграммы Гейна-Сарсона являются основой анализа требований и функционального проектирования информационных систем в пакете CASE.Аналитик фирмы MacroProject [54], а функциональные диаграммы виртуальных приборов пакета LabView позволяют создавать, отлаживать и исполнять прикладные программы, не используя ничего, кроме графических средств.

Таким образом, понятия «смысловое содержание» и «синтаксис» в функциональных диаграммах изначально являются очень тесно связанными, и для того чтобы разделить эти понятия и сформулировать, в чём состоит принципиальное отличие между ними, нам понадобится дополнительное обсуждение.

Пример 1.2.1. Один мой знакомый по ошибке включил радиоприёмник не в радиотрансляционную сеть, а в сетевую розетку ~ 220 в. Радиоприёмник сгорел.

Можно сказать, что в этом примере в роли «синтаксиса» соединения между прибором и розеткой выступает форма розетки и вилки, так что, с точки зрения синтаксиса, ошибочное подключение радиоприёмника, вызвавшее его повреждение, оказалось вполне «корректным». В то же время было нарушено смысловое содержание соединения — «приём радиопрограмм».

Рассмотренный пример позволяет наглядно показать отличие между синтаксисом и семантикой связей системы, и тем не менее, он является «вырожденным случаем». Дело в том, что случайности, подобные описанной, легко можно предотвратить, если использовать *разные* формы розеток для *разных* типов коммуникаций и таким образом гарантировать семантическую правильность любых возможных соединений чисто синтаксическими средствами. Такой принцип широко применяется в различных методах структурного анализа, проектирования и визуального программирования, использующих функциональные диаграммы — примерами использования этого принципа являются выделение типов связей между блоками, а также поддержка *словарей данных* [11], в которых со-

[2]Для описания динамики системы в SADT используются текстовые примечания, так называемые «свойства» и «действия» системы [19].

бирается информация о всех функциях и потоках данных проектируемой системы.

Как видим, отличие между синтаксисом и семантикой функциональных диаграмм является значительно более тонким, чем это может показаться на первый взгляд.

Пример 1.2.2. Когда в нашем институте стали проводить локальную сеть, очень быстро выяснилось, что прямое соединение компьютеров при помощи коаксиального кабеля является допустимым только в том случае, если они находятся в одном помещении и *подсоединены к одной точке заземления*. Во всех остальных случаях между компьютерами необходимо вводить *гальваническую развязку*, причём нарушение этого правила может привести к весьма печальным последствиям, вплоть до физического повреждения коммуникационных плат.

Этот пример является более интересным, поскольку «семантика» коаксиального соединения между компьютерами оказывается зависящей от того, в каком режиме используются соединяемые устройства и — что ещё более интересно — от состояния других связей (а именно, заземлений) этих устройств. Смысл одного и того же соединения, с одним и тем же «синтаксисом» меняется при изменении некоторых факторов, не имеющих к этому соединению прямого отношения.

Рассмотренный пример демонстрирует принципиальное отличие синтаксиса и семантики связей в информационной системе, состоящее в том, что *синтаксис* является статическим свойством, характеризующим некоторые конкретные соединения, *семантика* которых, в общем случае, определяется состоянием, режимом использования и свойствами различных компонентов системы или даже всей системы в целом.

Синтаксис связей в системе может быть выражен с помощью набора статических параметров — требований, предъявляемых к этим соединениям, а для описания семантики связей нам понадобится, в общем случае, набор логических правил, учитывающий структуру системы, а также режимы использования и свойства её компонентов. Это означает, что *для анализа семантической правильности функциональных диаграмм* совершенно недостаточно существующих (синтаксических) средств описания, и для решения этой задачи *необходима разработка более мощных средств, основанных на логике.*

Правда, осуществляя такую постановку задачи, необходимо ответить ещё на один принципиальный вопрос, а именно — каким же образом успешно «работают» существующие структурные методы, не распола-

гающие тонкими методами анализа правильности разрабатываемых информационных систем?

Отвечая на этот вопрос, необходимо обратить внимание на следующий факт, касающийся современных методов системотехнического проектирования [7]. Все эти методы, по своей сути, нацелены на разработку точной и полной спецификации (или нескольких вариантов спецификации) проектируемой системы, и поэтому *семантика* разрабатываемой системы формализуется и отражается в разрабатываемых функциональных диаграммах лишь в той мере, в которой она необходима на дальнейших этапах разработки системы, при этом бóльшая часть «семантического содержания» проекта, на основе которой осуществляются различные проектные решения, используется лишь в процессе проектирования, оставаясь «в головах проектировщиков»[3].

Вместе с тем, многие современные идеи и тенденции в области разработки и использования информационных систем явно не укладываются в «классические» представления о целях анализа и проектирования. Среди таких идей можно выделить следующие:

- *Концепция открытых информационных систем* [6, 32]. Несмотря на то, что существует большое количество точек зрения, что такое «открытая» информационная система, причём весьма авторитетных специалистов, я рискну предложить собственное определение «открытости». По существу, *открытой является такая информационная система, которая была спроектирована с учётом возможности её поддержки, наращивания и модификации (то есть дальнейшего развития) независимыми разработчиками на основе открытых*[4] *стандартов и спецификаций компонентов.*

  Обычно называют четыре аспекта открытости, а именно: *интероперабельность* информационных систем — возможность взаимодействия различных прикладных программ, реализованных, возможно, на разных аппаратных платформах; *функциональная расширяемость* — обеспечение развития функций конкретных систем при изменении требований к этим системам и, следовательно, сохранение сделанных ранее вложений; *переносимость программного обеспе-*

[3]Возможно, именно этим объясняется то, что некоторые разработчики предпочитают игнорировать формальные методы оптимизации проектируемой информационной системы [8].

[4]То есть общедоступных, в использовании и развитии которых могут принимать участие все заинтересованные стороны.

чения — возможность переноса программного обеспечения на новые аппаратные платформы; *мобильность персонала* — облегчение перехода пользователей от одной аппаратно-программной платформы к другой. Однако, как можно заметить, все эти аспекты сводятся к требованию обеспечения «гибкости» и развиваемости проектируемой системы, то есть приложения дополнительных усилий, чтобы обеспечить возможность пересмотра и дополнения в будущем принятых ранее проектных решений.

Таким образом, сущность «открытости» лежит в методологии проектирования, и концепция открытых информационных систем неизбежно приводит нас к необходимости более полного и точного представления (в графическом, формализованном или других видах) *семантики* проектируемой системы на всех этапах её жизненного цикла. Кроме того, для построения открытых информационных систем, как правило, используются компоненты с жёстко стандартизованными интерфейсами, так что анализ правильности таких систем может и должен осуществляться автоматически, с использованием формализованных правил построения соответствующих стандартных соединений.

- К перечисленным выше идеям тесно примыкает *объектно-ориентированный подход* к анализу, проектированию и программированию информационных систем [43, 4, 68, 69], который несёт в себе целый набор прогрессивных принципов, среди которых принцип выделения и многократного использования компонентов на основе *абстракции* и *наследования*, а также облегчение сопровождения и модификации информационной системы с помощью строгого разделения спецификаций и реализаций её компонентов и *локализации* используемых данных и механизмов их обработки. Развитие объектно-ориентированных подходов в информатике также приводит к идеям *семантической интероперабельности* [8] и использования формализованных методов для её обеспечения [52].

- То же самое можно сказать и о идее *повторного использования* компонентов информационных систем [68], которая, так же как и названные выше концепции, направлена на сохранение сделанных ранее вложений путём использования в новых разработках компонентов, созданных ранее в рамках других проектов.

- И конечно, *обеспечение участия компьютера в процессе проектирования информационной системы* (хотя бы в рамках некоторых CASE-средств на этапах анализа и проектирования) само по себе является задачей, требующей увеличения «семантического наполнения» средств спецификации — в том числе и функциональных диаграмм — применяемых в качестве языка общения человека и машины, для того чтобы машина могла принять на себя бóльшую часть работы проектировщика.

Характерным примером задачи, не укладывающейся в рамки «классических» представлений о целях проектирования, является *разработка профилей стандартов информационных систем* [6]. Дело в том, что *профиль стандарта*, по определению, является специально подобранным набором стандартов, понятийно и терминологически согласованных между собой и призванных обеспечить чёткие требования, в рамках которых могут быть спроектированы информационные системы некоторого определённого класса, имеющие, вообще говоря, неограниченное количество вариантов построения. Использование графических диаграмм может значительно облегчить понимание и использование разработанного профиля, однако при этом надо понимать, что отторжение и передача другому коллективу разработчиков диаграмм, созданных разработчиками профиля, обязательно должно сопровождаться передачей их семантического наполнения, которое можно было бы использовать при конкретизации требований и выборе вариантов построения информационной системы.

Интересно, что в приведённом примере достаточно чётко проявляется ещё одна роль, которую могут получить средства обеспечения семантической правильности функциональных диаграмм, выходящая за рамки собственно анализа и проектирования информационных систем. Это — *роль выразительного средства для описания и иллюстрации профилей стандартов открытых систем.* Такие средства (пока только гипотетически) могли бы представлять собой сочетание набора функциональных диаграмм и программ поддержки их семантической правильности, позволяющих читателю в интерактивном режиме собрать и конкретизировать различные варианты построения информационных систем и получить общие представления о взаимном влиянии различных требований, заложенных в профиле стандарта. Конечно, такой электронный документ не может рассматриваться в качестве обязательной составной части профиля стандарта, однако современный уровень развития глобальной информационной сети Internet позволяет говорить о реальной возможности

применения таких электронных публикаций.

Ещё одним важным следствием отказа от традиционных представлений о процессе проектирования является его разделение, по крайней мере, на два уровня, условно соответствующих этапам:

- разработки спецификаций компонентов информационных систем;
- сборки спецификаций конкретных информационных систем из спецификаций компонентов.

Эти два уровня, которые можно назвать «системным» и «пользовательским» (по аналогии со структурой систем, накапливающих и использующих знания), в частности, предъявляют разные требования к средствам представления и анализа семантики функциональных диаграмм, применяемых в процессе проектирования:

- На *системном уровне* наиболее важными качествами таких средств являются выразительная мощность, а также обеспечиваемые ими точность и полнота представления знаний о семантике диаграмм. По этой причине на системном уровне представляется вполне допустимым и даже желательным использование логических средств, конкретно, средств объектно-ориентированного логического программирования.
- В то же время, на *пользовательском уровне*, наоборот, является желательным сокрытие формализованных средств описания семантики и поддержка визуального интерактивного режима общения проектировщика с компьютером.

В настоящей работе основное внимание будет сосредоточено на первом — системном — уровне средств описания и анализа семантики функциональных диаграмм. Пользовательский уровень будет представлен в виде примеров использования различных видов *семантических моделей* информационных систем.

1.3 Проблемы логической интерпретации функциональных диаграмм

Основными проблемами, затрудняющими использование логических языков и логического программирования для поддержки интерактивного проектирования информационных систем, являются:

- Отсутствие вполне развитого и общепринятого подхода к интерпретации в логических языках объектов и связей между ними.
- Недостаточная теоретическая проработка вопросов взаимодействия человека и компьютера в логическом программировании.
- Проблема поддержки согласованного состояния объектов в моделях информационных систем.

Каждая из этих проблем соответствует отдельной области исследований, со своими задачами и методами их решения.

1.3.1 Логическая интерпретация объектов

Прежде всего, необходимо отметить, что, с точки зрения математической логики, исчисление предикатов первого порядка и основанные на нём логические языки являются ни чем иным как средством описания «объектов» и «связей» между ними, но при этом под словом «объект» понимается просто элемент данных — терм (например, имя человека или число), а формулы рассматриваются как описание связей между такими «объектами». Конечно, подобное понимание термина «объект» является несравнимо более бедным, чем «объекты» в процедурном ООП, несовместимым с такими идеями, как взаимодействие объектов, изменение состояния объекта, согласование объектов и т. п. Для решения поставленной задачи необходима более сложная логическая интерпретация понятий «объект» и «связь», более близкая по своим выразительным возможностям к понятиям ООП.

Проблема логической интерпретации понятий ООП имеет давнюю историю и обширную библиографию [46, 47, 71]. Одним из важнейших выводов, который можно сделать на основе анализа этого направления исследований, состоит в том, что *понятие «объект» процедурного ООП не имеет однозначной интерпретации в логике* и при попытке «перенести» его в логические языки распадается, по крайней мере, на три совершенно независимых аспекта, каждый со своими теоретическими задачами и средствами практической реализации:

- *Структурный аспект*, состоящий в том, что «объекты» в процедурном программировании являются естественным средством структурирования текста программы. Этому аспекту в логическом программировании и логических языках спецификаций соот-

ветствуют средства структурирования текста логической программы и, что более важно, средства управления пространством поиска [20, 63, 2, 34, 50, 5].

- *Динамический аспект*, отражающий возможности, связанные с изменением состояния объектов в процедурном ООП. Именно этот аспект вызывает наибольшие трудности в теории логического программирования и до сих пор не имеет универсальной интерпретации, пригодной для различных случаев использования логики [46, 53, 50, 59].

- *Информационный аспект*, связанный с проблемой описания сложных структур данных [40, 61, 75, 59].

Как это ни странно, именно проблема описания сложных структур данных, в качестве решения которой объектно-ориентированный подход пришёл в процедурное программирование, оказалась в наименьшей степени затронутой результатами, полученными в области объектно-ориентированного логического программирования. Названные выше проблемы структурирования пространства поиска и изменения состояний объектов оказались, по существу, никак не связанными с проблемой описания данных в логических языках, и, несмотря на попытки многих авторов свести воедино различные аспекты объектно-ориентированного подхода на основе некоторого универсального синтаксиса (см., например, [75, 59]; не совсем удачными подходами, ведущими к потере логической семантики разрабатываемых языков, можно назвать [79, 62, 18, 78, 57, 74, 48, 81, 77]), следует, видимо, честно признать, что названные аспекты действительно соответствуют *разным* логическим понятиям.

С точки зрения проблемы описания объектов и связей моделируемой системы, наиболее существенными являются первый и третий из названных выше аспектов. Проблема изменения состояний объектов в логическом программировании относится, в большей степени, к задаче логической интерпретации человеко-машинного взаимодействия.

1.3.2 Взаимодействие человека и машины

Интересным фактом в истории логического программирования является то, что оно выросло из исследований в области общения человека с машиной в искусственном интеллекте — первая реализация языка Пролог

была создана в марсельской группе под руководством Колмероэ в качестве средства для анализа предложений на естественном языке [56] — однако при этом именно проблема взаимодействия человека и компьютера остаётся до сих пор одним из наиболее «тёмных» мест в теории и практике логического программирования.

Проблемы с логической интерпретацией человеко-машинного взаимодействия имеют весьма серьёзные причины, основной из которых является фундаментальное противоречие, существующее между понятием взаимодействия и декларативной семантикой логических программ. Дело в том, что программа, взаимодействующая с человеком, является частным случаем так называемых *реагирующих систем* (*reactive systems*), отвечающих на сообщения извне. Характерной особенностью реагирующей системы является её последовательный переход из одних состояний в другие под воздействием внешних событий. В то же время, каждая логическая программа имеет декларативную (теоретико-модельную) семантику и может быть представлена в виде формулы (обычно в логике предикатов первого порядка) — то есть является статической системой. На практике это фундаментальное ограничение выражается, как правило, в следующих требованиях к реализациям логических языков:

- необходимость восстановления состояния программы в случае её отката после неуспешной попытки построить некоторую ветвь доказательства;

- требование, чтобы доказательство одних и тех же предикатов с одними и теми же аргументами всегда завершалось одинаково — либо успехом, либо неудачей, причём с одними и теми же выходными данными.

Конечно, операции ввода-вывода, пришедшие в Пролог из процедурных языков, не соответствуют ни одному из названных требований.

Впрочем, наличие декларативной семантики у логических языков вовсе не является непреодолимым препятствием для имитации изменений состояния системы, взаимодействующей с пользователем. В процессе развития логического программирования было предложено большое количество способов имитации объектов с изменяемым состоянием [46, 61], основанных на использовании задержанных вычислений [35, 13], параллельных стратегий исполнения логических программ [42, 58, 73, 78, 57, 51], а также на использовании метасредств [50] и неклассических логик [53, 49, 72].

В настоящей работе для интерпретации человеко-машинного взаимодействия предложен иной подход [25], основанный на *повторных доказательствах подцелей логической программы*, отличающийся от предложенных ранее наличием следующих достоинств:

- В основе подхода лежит использование классической логики (логики предикатов первого порядка).
- Центральной идеей и сущностью подхода является обнаружение и устранение логических противоречий, возникающих в процессе взаимодействия объектов.
- Разработанный подход позволяет корректным образом интерпретировать в логическом языке *разрушающее присваивание* и *необратимые процессы*, и на этой основе — визуальный (ориентированный на обработку событий) человеко-машинный интерфейс.

Таким образом, метод повторного доказательства, предложенный в данной работе, играет также ключевую роль в решении проблемы логической интерпретации взаимодействия и согласования объектов.

1.3.3 Поддержка согласованности объектов

Поддержка согласованности компонентов функциональной диаграммы является необходимым условием обеспечения её семантической правильности, а именно — выявления и устранения противоречий, возникающих в процессе интерактивного изменения диаграммы человеком.

Проблема поддержки согласованности объектов давно изучается в области *децентрализованного искусственного интеллекта* [80] и нашла отражение в таких подходах как открытые системы искусственного интеллекта [66, 37, 44], системы поддержки истинности [67, 80, 55], ограничения целостности баз данных [45] и т. п. В логическом программировании эта проблема обычно не ставится в явном виде и решается с помощью имитации таких процедурных средств как демоны [79]. Следует, однако, отметить работу [74], в которой проблема согласования объектов сформулирована в явном виде, хотя при этом её решение не является логическим, так как оно игнорирует декларативную семантику языка.

В настоящей работе проблема поддержки согласованности объектов решается с помощью чисто логических средств, основанных на методе повторного доказательства подцелей логической программы.

1.4 Выводы

Современные тенденции развития информатики, такие как *открытые информационные системы*, *объектно-ориентированный подход к анализу, проектированию и программированию*, *повторное использование компонентов* создают предпосылки для разработки новых методов и средств структурного системного анализа, проектирования и визуального программирования, обеспечивающих *семантическую правильность* проектируемых диаграмм.

Перспективным подходом к разработке и реализации таких методов и средств является сочетание графических диаграмм для поддержки визуального интерфейса «человек-машина» и средств объектно-ориентированного логического программирования, позволяющих «оживить» диаграммы с помощью формализованного описания и анализа их семантики в процессе интерактивного проектирования.

Глава 2

Метод интерактивного функционального моделирования информационных систем и Акторный Пролог

Для анализа и обеспечения семантической правильности функциональных диаграмм в настоящей работе предлагается *метод интерактивного функционального моделирования*, предусматривающий:

- осуществление *семантического анализа* (анализа смысла) функциональных диаграмм;
- выявление и устранение смысловых противоречий в процессе интерактивного редактирования диаграммы проектировщиком.

Метод интерактивного функционального моделирования основан на трёх идеях, которые можно условно назвать:

- «логика»: использование логической интерпретации функциональных диаграмм и средств логического анализа диаграмм;
- «интерактивность»: интерактивный режим построения и использования функциональных диаграмм;

- «акторы»: взаимодействие и согласование компонентов функциональной диаграммы в процессе её интерактивного редактирования.

Конкретно, предлагаемый метод включает следующий план действий:

1. Одновременно с разработкой ФД информационной системы осуществляется построение программы на специальном логическом языке, описывающей блоки и связи ФД в терминах логического программирования. Можно выделить, по крайней мере, три разновидности таких логических программ — мы будем называть их *семантическими моделями*:

   (a) семантические модели структуры ФД информационной системы — *структурные семантические модели*, предназначенные для проверки и логического доказательства правильности соединения компонентов, а также анализа свойств системы;

   (b) *интерактивные семантические модели*, обеспечивающие семантическую правильность ФД проектируемой системы в процессе её интерактивного редактирования;

   (c) *имитационные семантические модели* для моделирования работы системы, которые могут также рассматриваться в качестве парадигмы объектно-ориентированного логического программирования информационных систем.

   Построение семантической модели может происходить частично автоматически, с помощью трансляции разрабатываемой ФД, а частично — «вручную», путём формализации знаний экспертов о семантике составных частей диаграммы.

2. Начиная с некоторого момента, когда накоплено достаточно информации о блоках и связях, а также о правилах соединения блоков ФД (то есть построение семантической модели, хотя бы в первом приближении, завершено), построенная логическая программа может быть исполнена. В дальнейшем может быть осуществлён один из двух вариантов использования этой программы:

   (a) Программа исполняется время от времени для анализа заданных параметров информационной системы или проверки правильности соединения функциональных блоков. Такой режим использования лучше подходит, скорее, для структурных и имитационных семантических моделей.

(b) Программа реализует интерактивный режим работы проектировщика с машиной, реагируя на изменения, вносимые им в структуру ФД или параметры её составных частей, и проверяя правильность соединения и согласованность режимов работы компонентов моделируемой системы.

3. Семантическая модель является «отторгаемым продуктом» и может быть передана другому проектировщику, при этом логические правила соединения блоков могут быть сокрыты в описании компонентов ФД, что позволяет использовать их как «чёрные ящики». Более того, могут быть отдельно переданы компоненты семантической модели, разработанные «вручную», что позволит другому проектировщику автоматически (с помощью трансляции своих ФД) строить свои собственные семантические модели.

Для построения и исполнения различных видов семантических моделей в диссертации разработан Акторный Пролог [26, 22] — объектно-ориентированный логический язык, реализующий обобщённую *логическую акторную модель ФД информационной системы, проектируемой в интерактивном режиме*, которая будет рассмотрена ниже.

Примеры построения и использования семантических моделей будут приведены далее в этой главе, а также в главе «Использование Акторного Пролога для логического анализа функциональных диаграмм».

2.1 Логическая акторная модель ФД информационных систем

В качестве обобщённой модели *ФД информационной системы, проектируемой в интерактивном режиме*, разработана *логическая акторная модель* [23] (см. рис. 2.1), представляющая диаграмму в виде некоторой теоремы на логическом языке, разделённой на *логические акторы*[1] — *повторно доказываемые* подцели ($\alpha_1, \ldots, \alpha_n$ на рисунке), взаимодействующие через *общие переменные* ($V_1, \ldots, V_m$). Доказательство этой теоремы осуществляется в некотором *объектно-ориентированном пространстве поиска*, состоящем из отдельных миров ($W_1, \ldots, W_k$), топология которого соответствует структуре ФД моделируемой системы.

[1]Такое название выбрано потому, что отправной точкой для разработки модели послужила акторная модель вычислений Хьюитта [66].

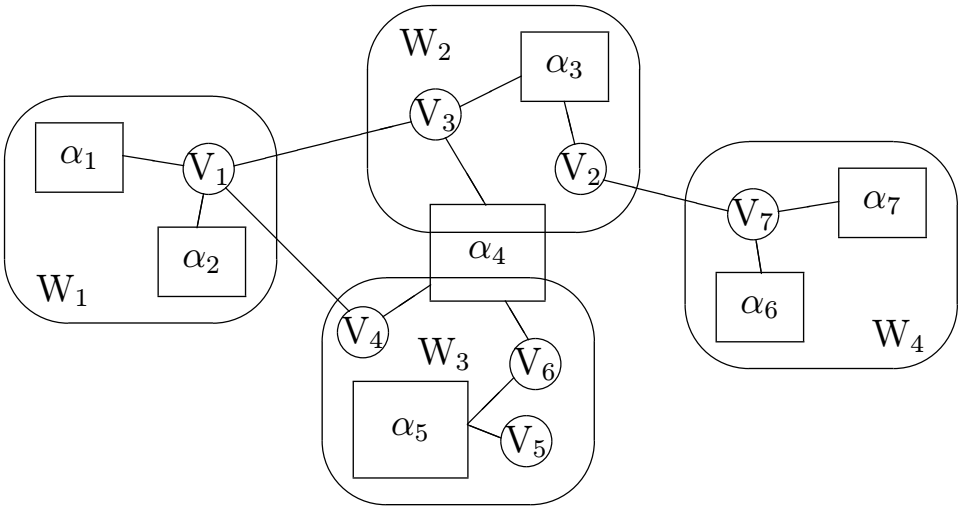


Рис. 2.1: Логическая акторная модель ФД.

С точки зрения логики, взаимодействие человека с компьютером в процессе интерактивного построения и анализа ФД удобно рассматривать как доказательство некоторой теоремы о свойствах проектируемой системы, в котором одновременно принимают участие человек, изменяющий условия задачи, и машина, обеспечивающая корректность доказательства, приводя его по мере необходимости в соответствие с изменяемыми исходными данными.

Логическая акторная модель вычислений, соответствующая такому представлению ФД информационной системы, основана на *повторных доказательствах акторов*, осуществляемых при изменении значений общих переменных, в целях обеспечения *корректности и полноты доказательства теоремы*.

Взаимодействие между акторами реализуется с помощью операции «корректного разрушающего присваивания», изменяющей значения общих переменных программы. В ходе этой операции (например, в процессе доказательства специального предиката корректного разрушающего присваивания *':='*) осуществляется необходимое изменение значений общих переменных, а затем вызывается повторное доказательство акторов, зависящих от старых значений общих переменных. Если повторное доказательство всех этих акторов завершается успехом, исполнение операции также заканчивается успехом (в этом случае предикат разрушающего присваивания считается истинным, и его доказательство завершается успехом), в противном случае в логической программе происходит откат.

В рамках предложенной модели изменение человеком условий задачи

во время логического доказательства интерпретируется как использование им корректного разрушающего присваивания, вызывающего повторное доказательство акторов логической программы. Соответствующая стратегия управления, обеспечивающая корректность доказательства теоремы при изменении значений общих переменных и произвольном порядке доказательства акторов, названа *механизмом повторного доказательства* (*акторным механизмом*).

Следует отметить, что хотя в качестве исходной идеи для разработки логической акторной модели ФД информационных систем послужила акторная модель вычислений Хьюитта [66, 44], однако, в отличие от «классической» акторной модели, логическая акторная модель вычислений обеспечивает иной принцип взаимодействия акторов, поскольку обмен информацией между логическими акторами осуществляется только через общие переменные.

В настоящей работе в качестве языка, реализующего логическую акторную модель вычислений, разработан логический язык программирования — Акторный Пролог, в котором воплощена логическая интерпретация понятий ООП, отражающая его *структурные*, *динамические* и *информационные* аспекты. В Акторном Прологе перечисленные аспекты ООП реализованы с помощью:

- классов, миров и наследования;
- акторного механизма;
- простых и составных термов.

Использование названных средств языка в процессе функционального моделирования основано на следующих принципах [24]:

1. Каждый функциональный блок диаграммы описывается *экземпляром некоторого класса* логического языка, содержимым которого являются правила, описывающие семантику блока.

2. Связи между блоками функциональной диаграммы описываются с помощью *слотов* экземпляров классов. Для этих же целей удобно использовать *правила второго порядка* [26], имитируемые в Акторном Прологе с помощью *недоопределённых множеств* [26, 22].

3. *Логические акторы* и *механизм повторного доказательства* служат для поддержки интерактивного проектирования функциональных диаграмм.

Модель системы, собранная из отдельных классов, обладает как *операционной*, так и *декларативной семантикой* и может быть однозначно выражена в терминах логики предикатов первого порядка, что позволяет обеспечивать *логическую корректность* проводимого анализа диаграмм. Вторым важным свойством семантических моделей является возможность обеспечивать *логическую полноту* проводимого анализа. Другими словами, если моделируемая система обладает некоторым (возможно, неблагоприятным) свойством, обнаружение этого свойства гарантируется, хотя для этого, возможно, придётся проанализировать большое количество режимов её работы.

Прежде чем перейти к описанию конкретных разновидностей семантических моделей и средств логического языка, необходимо сделать следующее замечание. Точно так же как чистый Пролог является всего лишь иной формой записи некоторых формул логики предикатов первого порядка, все новые синтаксические конструкции, вводимые в настоящей работе, являются видоизменением чистого Пролога и, в этом смысле, также эквивалентны формулам логики предикатов.

Последовательно придерживаясь такого подхода, мы *не можем* просто «ввести» в чистый Пролог какие-то новые понятия и математические объекты, так как это означало бы построение некоторой новой логики, отличной от классической, и, скорее всего, потерявшей часть её полезных свойств. Вместо этого в данной работе вводятся некоторые новые формы записи формул, напоминающие по своему внешнему виду и способам применения некоторые средства процедурного программирования — акторы, абстрактные типы данных, недоопределённые множества и др. — но при этом имеющие совсем другое происхождение и другой смысл. Таким образом, следует помнить, что понятия «актор», «класс», «экземпляр класса» и т. п., используемые в данной работе, являются — так же как и многие другие понятия в логическом программировании — не более чем логическими аналогами соответствующих понятий процедурного программирования.

2.2 Структурные семантические модели

Простейшими семантическими моделями, соответствующими понятию «топология пространства поиска» в рассмотренной модели ФД информационных систем, являются *структурные* семантические модели.

Структурными семантическими моделями мы будем называть логические программы, описывающие каждый блок ФД информационной системы в виде некоторого объекта логического языка (точнее, *экземпляра класса* или «мира», в терминах Акторного Пролога).

Отметим, что распространённые в настоящее время структурные методы, использующие функциональные диаграммы, а также соответствующие CASE-системы не предусматривают выделение классов функциональных блоков — предложенный в диссертационной работе *механизм классов* логического языка обеспечивает новые возможности для спецификации блоков функциональных диаграмм на основе *абстракции*, *наследования* и *повторного использования* описаний функций информационных систем.

Механизм наследования, реализованный в логическом языке, позволяет использовать принцип абстракции при разработке описания модулей информационных систем и создавать спецификации модулей на основе библиотек спецификаций стандартизованных модулей.

2.2.1 Классы, миры, наследование

В Акторном Прологе для формирования топологии пространства поиска используется *механизм классов*.

По аналогии с классами в процедурном ООП, в Акторном Прологе введены синтаксические конструкции для структурирования текста программы — классы. *Классом* в языке называется набор логических предложений (фактов, правил). Так же как в процедурном программировании, каждому классу соответствует уникальное имя, и все они являются элементами некоторой *иерархии наследования*.

Понятию «экземпляр класса» процедурного программирования в Акторном Прологе также соответствует аналог. *Экземплярами классов* («мирами») в Акторном Прологе называются конкретные применения классов, однако создаются они не в результате выполнения процедурной операции *new* (как, например, в C++ [31]), а в результате доказательства некоторых специальных формул, названных «конструкторами». При этом логическая сущность экземпляров классов в Акторном Прологе приводит к одному весьма существенному отличию от их процедурных аналогов, а именно к тому, что в логическом языке экземпляры классов удаляются в процессе отката программы — точно так же как любые другие результаты логического вывода. Никаких аналогов процедурной операции *delete* и понятия «деструктор» в Акторном Прологе нет, так

как в логическом языке не может быть процедурных операций явного удаления экземпляров классов.

Ещё одной важной особенностью Акторного Пролога является то, что в этом языке (в отличие от процедурных объектно-ориентированных языков типа Smalltalk [28, 34] и C++) осуществлено последовательное разделение понятий «экземпляр класса» и «элемент данных»[2]. То что эти понятия действительно соответствуют двум совершенно разным сущностям, видно, по крайней мере, из следующих сопоставлений:

1. В логическом языке переменная может быть как связанной (имеющей значение), так и «неопределённой», и, в общем случае, нет никакой возможности угадать заранее (во время трансляции программы), в каком состоянии она окажется. Это значит, что невозможно построить «чистый» логический язык, в котором обращение к значению переменной («вызов метода объекта», в терминах процедурных языков) было бы корректной, всегда определённой операцией, не прибегая к различным ухищрениям с задержанными вычислениями [35, 13].

   В процедурных языках подобные проблемы не возникают, так как значение любой переменной в них можно считать всегда известным (в крайнем случае, она может иметь начальное значение *nil*), однако в логическом программировании названное противоречие приводит к тому, что экземпляры классов становится удобнее хранить с помощью специальной системы обозначений, существующей независимо от переменных и аргументов процедур. В Акторном Прологе такой системой обозначений являются *слоты* экземпляров классов, о которых будет рассказано ниже.

2. В объектно-ориентированном языке удобно придерживаться *принципа уникальности экземпляров классов*, то есть рассматривать все экземпляры классов (даже соответствующие одному классу) как разные сущности, которые, в частности, не могут быть друг с другом унифицированы. Этот принцип не позволяет рассматривать в качестве экземпляров каких-либо классов значения термов языка, так как это помешало бы осуществлять унификацию термов и разрушило бы весь механизм логического вывода.

[2]Именно по этой причине в Акторном Прологе экземпляры классов именуются не «объектами», а «мирами».

Экземпляры классов служат составными частями пространства поиска (*закрытыми блоками* [63]) во время исполнения программы. Содержимым экземпляра класса являются:

1. набор предложений соответствующего класса, а также предложений предков этого класса в иерархии наследования;

2. набор слотов, доступных из всех предложений экземпляра класса.

Механизм наследования в Акторном Прологе является аналогом подобных механизмов в процедурных объектно-ориентированных языках, однако он имеет несколько иную операционную семантику: иерархия наследования задаёт *правила поиска* [36] предложений программы, осуществляемого в процессе её исполнения. Например, если в качестве пространства поиска для доказательства некоторого предиката p служит экземпляр класса A, то поиск соответствующего предложения с заголовком p будет осуществляться по очереди среди предложений класса A, затем среди предложений непосредственного предка класса A и так далее, пока не будет найдено подходящее предложение, или пока не закончится иерархия наследования.

Таким образом, в Акторном Прологе не происходит *перекрытие методов* [38], хотя его можно смоделировать, используя вне-логический оператор отсечения '!'. Кроме того, в Акторном Прологе не используется так называемое *множественное наследование* [31, 38] — у класса может быть лишь один непосредственный предок, так как иначе было бы непонятно, в каком порядке следует просматривать предложения родительских классов в ходе логического вывода, и пришлось бы назначать этот порядок искусственно, с помощью волевого решения.

Слот — это «глобальная переменная» экземпляра класса или обозначение некоторого мира. Имена слотов названы *атрибутами классов*.

Атрибуты необходимо объявлять во всех классах, в которых используются соответствующие слоты. В объявлении атрибутов могут быть заданы *инициализаторы*, определяющие значения слотов, — термы языка, конструкторы или другие атрибуты.

Пример определения класса:

```
class 'ADDER' is
a
b    = 0        -- Определение атрибутов класса.
c1   = 0        -- Слоты b и c1 по умолчанию
```

```
sum             -- содержат 0.
c2
[               -- Предложения класса.
table(0,0,F, F,0).      table(0,1,0, 1,0).
table(0,1,1, 0,1).      table(1,0,0, 1,0).
table(1,0,1, 0,1).      table(1,1,F, F,1).
get_state(sum).
goal:-
    table(a,b,c1,sum,c2).
]
```

Класс *'ADDER'* изображает полный двоичный сумматор. При этом атрибуты a, b и $c1$ обозначают слагаемые сумматора и входной бит переноса, sum и $c2$ — сумму и выходной бит переноса.

Доказательство конструктора (*построение экземпляра некоторого класса* C) включает следующие этапы:

1. Формирование экземпляра класса. На этом этапе осуществляется:

   (a) Построение соответствующего пространства поиска.

   В состав пространства поиска входят предложения самого класса C, а также (в соответствии с иерархией наследования) предложения всех его предков.

   (b) Формирование слотов экземпляра класса.

   Одновременно с созданием каждого слота, если для него задан соответствующий инициализатор, формируется его «начальное» значение — элемент данных или мир. Слоты, не получившие значения в процессе своего создания, получают в качестве начальных значений некоторые уникальные общие переменные.

2. Доказательство предиката *goal*() во всех мирах, сформированных в процессе построения экземпляра класса C — доказательство конструктора будет считаться успешным, если во всех этих мирах доказательство предиката *goal* завершится успехом.

В нашем примере конструктор вида

('ADDER', a= 1, b= 1, sum= Result)

создаст новый экземпляр класса *'ADDER'* и присвоит значение переменной $Result = 0$.

Существенной особенностью языка является то, что в любой подцели предложения может быть явным образом (с помощью атрибута или конструктора) указан мир, в котором должно быть выполнено доказательство соответствующего предиката — такие подцели названы «дальними» вызовами предикатов. Если обозначением мира в подцели предложения является конструктор, доказательство этого конструктора происходит непосредственно перед вызовом предиката, всякий раз, когда осуществляется доказательство подцели. Таким образом, в процессе исполнения программы пространство поиска может развёртываться динамически.

Этот механизм языка, названный «динамическими мирами», можно проиллюстрировать следующим примером.

```
class 'PARALLEL_ADDER' is
input1   -- В этом примере значениями слотов
input2   -- экземпляра класса служат конечные
output   -- списки целых чисел.
[
goal:-
     loop(input1,input2,0,output).
loop([ ],[ ],_,[ ]):-!.
loop([A|R1],[B|R2],C1,[S|R3]):-
     ('ADDER', a= A, b= B, c1= C1, c2= C2) ? get_state(S),
     loop(R1,R2,C2,R3).
]
```

В процессе построения экземпляра класса *'PARALLEL_ADDER'* происходит динамическая сборка параллельного сумматора. Компонентами устройства служат экземпляры класса *'ADDER'*, создаваемые в ходе доказательства предиката *loop*: конструктор экземпляра класса *'ADDER'* явным образом определяет миры, в которых осуществляются дальние вызовы предиката *get_state*. Использование динамических миров в этом примере позволяет создавать сумматор любой необходимой разрядности, в зависимости от длины слов, подаваемых на вход устройства.

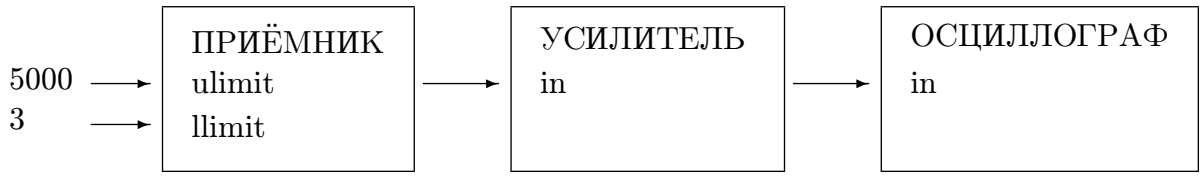


Рис. 2.2: Функциональная диаграмма измерительной системы.

2.2.2 Пример структурной семантической модели

На рис. 2.2 приведён пример функциональной диаграммы измерительной [1] информационной системы, в состав которой входят приёмник сигналов, усилитель и осциллограф.

Каждый компонент установки характеризуется некоторым собственным частотным диапазоном (или набором диапазонов), частотный диапазон всей системы в целом может быть выражен, исходя из параметров её компонентов:

```
class 'ПРИЁМНИК' is
ulimit    = 100.0                    -- Значения слотов
llimit    = 10.0                     -- по умолчанию.
[
range(llimit, ulimit).
goal.
]
-----------------------------------------------
class 'ПРИБОР' using 'ALPHA' is
-- Класс 'ПРИБОР' является потомком предопределённого
-- класса 'ALPHA', в котором реализованы некоторые
-- арифметические, управляющие предикаты и т. п.
in                                    -- Входной разъём.
[
range(Ymin, Ymax):-
     own_range(Smin, Smax),
     in ? range(Imin, Imax),          -- Дальний вызов предиката.
     max(Imin, Smin, Ymin),
     min(Imax, Smax, Ymax).
goal:- !.
```

```
max(Y1, Y2, Y1):- Y1 >= Y2,!.
max(_, Y2, Y2).
min(Y1, Y2, Y1):- Y1 <= Y2,!.
min(_, Y2, Y2).
]
```

```
class 'УСИЛИТЕЛЬ' using 'ПРИБОР' is
[
own_range(10.0, 100000.0).
]
```

```
class 'ОСЦИЛЛОГРАФ' using 'ПРИБОР' is
output = ('WINDOW', x= 15, y= 10, dx= 50, dy= 10)
-- Значение слота output строится в результате доказательства
-- конструктора экземпляра предопределённого класса 'WINDOW',
-- реализующего управление экранными окнами.
[
own_range(10.0, 1000.0).
own_range(100.0, 10000.0).
own_range(1000.0, 100000.0).
                    -- Предикат goal автоматически вызывается
goal:-              -- во время построения экземпляра класса.
    range(Ymin, Ymax),
    output ? write("Range=", Ymin," - ", Ymax, "\n"),
    fail.
]
```

```
project is
    ('ОСЦИЛЛОГРАФ',
                in= ('УСИЛИТЕЛЬ',
                            in= ('ПРИЁМНИК',
                                        ulimit= 5000.0,
                                        llimit= 3.0)))
-- Так обозначается целевое утверждение («проект») программы.
```

В ходе исполнения программы будет найден набор частотных диапазонов, в которых может работать проектируемая установка при различ-

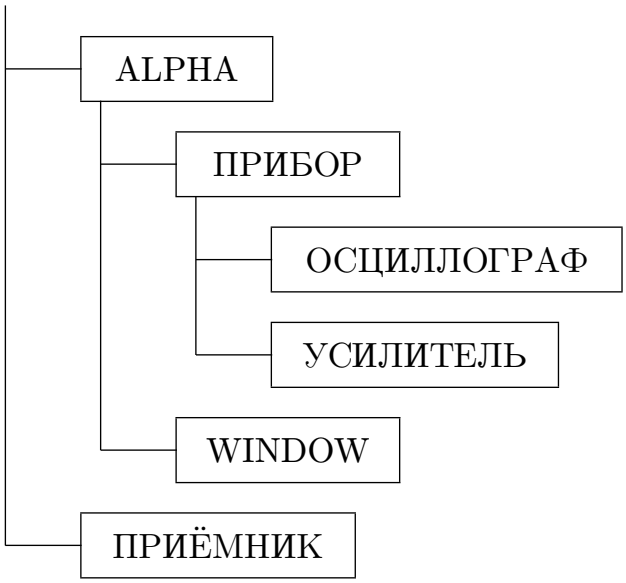


Рис. 2.3: Иерархия наследования.

ных положениях переключателей:

$$\text{Range} = [10..1000], [100..5e3], [1000..5e3].$$

Обратите внимание, что класс *'ПРИБОР'* в рассмотренной структурной модели является абстракцией свойств блоков «УСИЛИТЕЛЬ» и «ОСЦИЛЛОГРАФ», а соответствующие классы *'УСИЛИТЕЛЬ'* и *'ОСЦИЛЛОГРАФ'* являются потомками этого класса в иерархии наследования (рис. 2.3).

2.3 Интерактивные модели

Для реализации интерактивного режима проектирования, в котором проектные решения вырабатываются в результате совместной деятельности человека и машины, требуются более сложные — *интерактивные* семантические модели.

Интерактивными семантическими моделями мы будем называть такие модели, которые обеспечивают возможность изменять отдельные параметры ФД проектируемой системы или вносить изменения в её структуру и при этом отслеживают логическую согласованность её компонентов и корректность построения системы в целом. В терминах логической

акторной модели ФД информационных систем, это означает, что нам понадобятся логические программы, реализующие построение и взаимодействие логических акторов.

В Акторном Прологе разработаны *средства динамического определения акторов*, а также *акторный механизм*, управляющий повторным доказательством акторов.

Определение акторов осуществляется с помощью специального префикса `@`, действие которого можно проиллюстрировать следующим примером:

```
goal:-
    subgoal_a(X),
    subgoal_b,
    user_input(Y),
    X := Y.
subgoal_a(1).
subgoal_a(3).
subgoal_a(5).
(a)
```

```
goal:-
    @ subgoal_a(X),
    subgoal_b,
    user_input(Y),
    X := Y.
subgoal_a(1).
subgoal_a(3).
subgoal_a(5).
(b)
```

В роли предиката *user_input* в реальной задаче могли бы использоваться операция считывания значения поля экранного диалога, обращение к базе данных и т. п.

В случае (a) акторный механизм не используется. В процессе доказательства целевого утверждения *goal* сначала будет найдено решение $subgoal\_a(1)$, переменная X при этом получит значение 1. Затем будут исполнены подцели $subgoal\_b$ и $user\_input(Y)$ (пользователь введёт значение переменной Y, например, $Y = 5$). Когда очередь дойдёт до отношения $X := Y$, то окажется, что оно ложно: $1 \neq 5$. Это вызовет откат программы до последней точки ветвления — ей соответствует выбор нового факта для подцели $subgoal\_a(X)$. Следующая попытка доказательства (при $X = 3$) точно так же потерпит неудачу. И только с третьего раза (когда X получит значение 5) программа завершится успехом. Заметим, что подцели $subgoal\_b$ и $user\_input(Y)$ пришлось передоказывать по три раза, хотя они не имеют к переменной X никакого отношения.

В случае (b) подцель доказательства, соответствующая вызову предиката $subgoal\_a(X)$, будет объявлена актором. Следовательно, первая же попытка выполнить $X := 5$ завершится успехом. Последствием это-

го присваивания станет отмена результатов доказательства подцели $subgoal\_a(X)$ — «нейтрализация» актора — и её повторное исполнение (на этот раз будет выбран факт $subgoal\_a(5)$, соответствующий значению $X = 5$).

Заметим, что, в реальной задаче, повторное доказательство любой акторной подцели может «затронуть» некоторые другие акторы, что, в общем случае, вызовет лавинообразный процесс устранения противоречий, возникших в системе.

Декларативная семантика программы на Акторном Прологе без средств управления[3] принята равной декларативной семантике соответствующей программы без акторных префиксов.

Встроенный предикат *':='* языка реализует операцию разрушающего присваивания, изменяющую значения общих переменных. Декларативная семантика этого предиката точно такая же, как у обычного равенства в чистом Прологе, в то же время операционная семантика этого предиката определяется логической акторной моделью вычислений.

Заметим, что акторный механизм языка ни в коем случае *не следует* рассматривать в качестве разновидности отката (или интеллектуального отката) программы: при нейтрализации и повторном доказательстве актора старые результаты его доказательства не выбрасываются из стека (как это происходит при откате), а просто становятся временно недоступными и, в частности, могут стать снова «действующими», если в программе произойдёт настоящий откат, и результаты нейтрализации и повторного доказательства окажутся выброшенными из стека[4].

Необходимо также отметить, что нейтрализация акторов, сама по себе, никогда *не может привести к зацикливанию программы* — чтобы предотвратить такую опасность, в языке введён запрет на нейтрализацию акторов, доказательство которых ещё не окончено.

Отправной точкой для дальнейшего обсуждения нам послужит тот факт, что результатом успешного доказательства теоремы, в дополнение к вычисленным данным, является набор (сеть) согласованных между собой акторов. В этом, наверное, не было бы ничего интересного, если бы разделение теоремы на части осуществлялось статически. В Акторном Прологе, однако, используется динамическое определение акторов, и,

[3]Средствами управления Акторного Пролога являются отсечение '!' и другие встроенные операторы, заголовки внешних предложений, а также специальный акторный префикс «&».

[4]То есть акторный механизм можно назвать, скорее, «*анти*-бэктрэкингом».

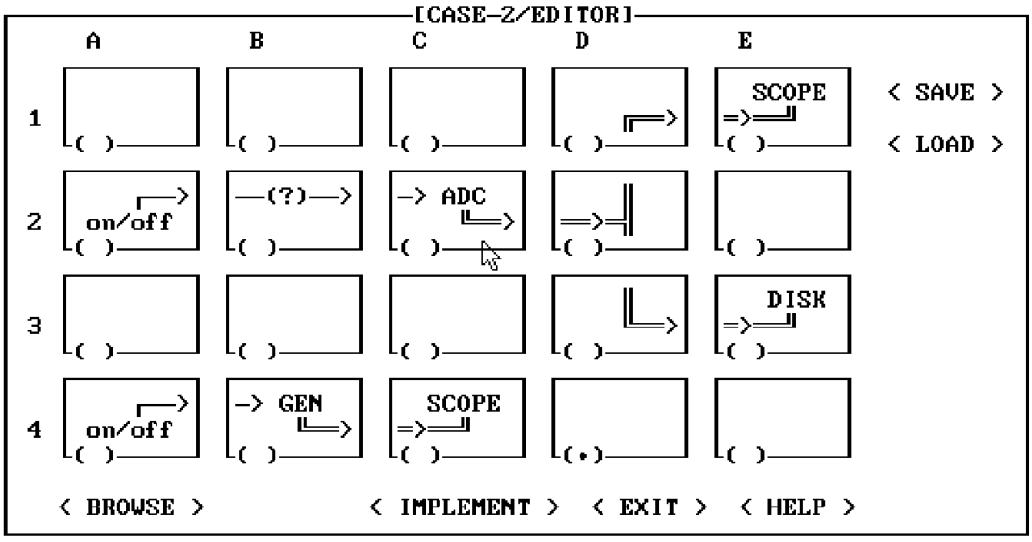


Рис. 2.4: Пример визуального интерфейса интерактивной модели.

следовательно, в результате вызванного пользователем повторного доказательства акторов, могут быть созданы новые акторы, то есть результатом доказательства теоремы может стать некоторая *новая программа* (новая теорема), целевому утверждению которой будет соответствовать другая формула логики предикатов.

Чтобы эта программа заработала, достаточно каким-либо образом (например, изменив значение одной из общих переменных) вывести её акторы из согласованного состояния. Разумеется, такая операция является отдельной «постановкой задачи» и не имеет никакого отношения к тому доказательству, результатом которого стала новая теорема. Кто именно запустит новую программу — совершенно не важно. Это может быть пользователь (на основе такого подхода осуществляется управление немодальными диалогами), ограничение целостности базы данных, таймер или какое-нибудь внешнее устройство.

На рис. 2.4 приведён пример визуального интерфейса логической программы (интерактивной семантической модели), помогающей конструировать ФД систем автоматизации физического эксперимента из некоторых готовых компонентов. Каждому блоку в составе диаграммы ставится в соответствие логический актор, оценивающий правильность подключения окружающих его компонентов. В случае несогласия с выбранным проектным решением, актор автоматически перекрашивает соответству-

ющий блок в фиолетовый цвет, в случае согласия — в зелёный.

Другой пример интерактивной семантической модели будет подробно рассмотрен в главе «Использование Акторного Пролога для логического анализа функциональных диаграмм».

2.4 Имитационные семантические модели

Эксперименты с реализацией Акторного Пролога показали возможность его использования не только для интерактивного функционального моделирования, но также для *имитационного моделирования* и *программирования* информационных систем. Преимуществом использования логического языка для этих целей является наличие у него строгой декларативной семантики, инвариантной по отношению к возможному недетерминизму поведения системы [21].

Наиболее интересной темой для обсуждения в связи с таким применением логического программирования является сочетание в логическом языке акторного механизма и механизма классов. Чтобы сформулировать *парадигму логического акторного программирования*, основанную на этих средствах, можно использовать следующую аналогию:

> пространство поиска и сеть логических акторов программы формируются подобно тому, как из микросхем собирается цифровое устройство, затем на «устройство» подаются «электрические сигналы» (изменение значений общих переменных это внешнее событие, на которое «устройство» должно реагировать) — и начинается переключение акторов.

Отличие состоит лишь в том, что пространство поиска и акторная сеть могут развёртываться динамически, во время исполнения программы — «выключать прибор из розетки» не надо. Как видим, это достаточно мощная парадигма программирования, и реализовать её не удалось бы иначе, как на стыке логического и объектно-ориентированного программирования.

В качестве примера использования акторного механизма для моделирования сетей взаимодействующих агентов, мы рассмотрим имитационную семантическую модель ФД виртуального прибора (см. рис. 2.5), созданного с помощью демонстрационной версии пакета LabView [30, 60].

В соответствии с алгоритмом работы, заданным функциональной диаграммой на рис. 2.6, виртуальный прибор осуществляет ввод зна-

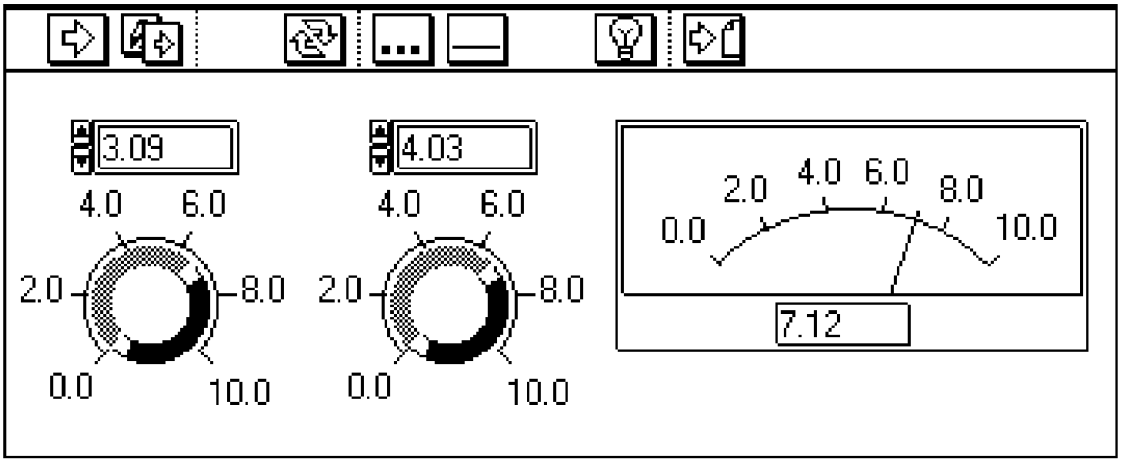


Рис. 2.5: Передняя панель виртуального прибора.

чений $INPUT1$ и $INPUT2$, задаваемых на его передней панели, суммирует эти значения и выводит полученный результат $OUTPUT$ обратно на переднюю панель.

Разработку модели мы начнём с описания виртуального устройства отображения $OUTPUT$ — определим класс *'INDICATOR'* (рис. 2.7).

В процессе создания экземпляра класса *'INDICATOR'*, соответствующего некоторому конкретному виртуальному устройству (в нашем случае — $OUTPUT$), вызывается определённый в его составе предикат *goal*, и происходит построение некоторого актора P, соответствующего акторному вызову предиката *control_state*. Этот актор с помощью встроенного предиката *in* получает значение общей переменной *state* и с помощью оператора *put* мира *panel* отображает это значение на экране. На этом доказательство предиката *goal* завершается, однако актор P остаётся зависящим от переменной *state* и будет в дальнейшем реагировать на все изменения значения этой переменной — в случае изменения значения *state*, актор P будет доказан повторно и выведет на экран новое значение *state*.

При построении модели виртуального устройства $INPUT$ для ввода данных, мы будем исходить из предположения, что для получения информации программа должна через определённые промежутки времени проверять состояние некоторого внешнего элемента данных (в нашей

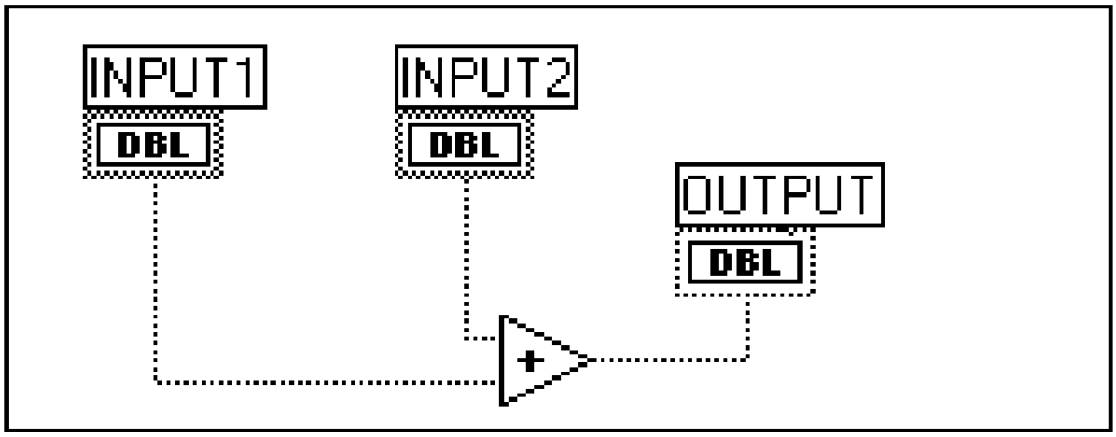


Рис. 2.6: Функциональная диаграмма виртуального прибора.

```
class 'INDICATOR' is
state
panel
entry
[
goal:-
    @ control_state.          -- Построение актора.
--
control_state:-               -- Вывод значения state текстового
    in(state),                -- поля entry экранного диалога.
    check_state(state),
    panel ? put(entry,state).
--
check_state(0.0):-!.
check_state(_).
]
```

Рис. 2.7: Определение класса 'INDICATOR'.

модели — поля редактирования экранного диалога) и присваивать его значение некоторой общей переменной.

Для реализации такого алгоритма работы виртуального устройства *INPUT* мы определим класс *'CONTROL'* (рис. 2.8). Класс *'CONTROL'* является потомком предопределённого класса *'TIMER'*, реализующего взаимодействие с системным таймером компьютера. В общем случае, в программе может существовать несколько «виртуальных таймеров» (экземпляров класса *'TIMER'*), имитирующих параллельное исполнение фрагментов программы. Они не вмешиваются в текущее доказательство теоремы и ждут его успешного окончания. После завершения логического вывода, если он окончится удачно, виртуальные таймеры становятся «постановщиками новых задач»: через некоторые конечные промежутки времени в них автоматически вызываются предикаты *tick* — считается, что миру класса *'TIMER'* соответствует некоторый актор, реагирующий вызовом предиката *tick* на изменение некоторой общей переменной 'системное время'.

Теперь мы можем перейти к описанию основного класса нашей модели — класса *'MODEL'* (рис. 2.9). Класс *'MODEL'* является потомком предопределённого класса *'DIALOG'*, управляющего экранными диалогами. В классе *'DIALOG'*, в частности, реализованы операторы ввода-вывода *get* и *put*, которые мы использовали при определении классов *'INDICATOR'* и *'CONTROL'*. Экземпляр класса *'MODEL'* включает три экземпляра вспомогательных классов, определённых ранее — два экземпляра класса *'CONTROL'* и один — класса *'INDICATOR'*, а также актор, соответствующий вызову предиката $sum(X, Y)$, который устанавливает логическую связь между состоянием переключателей и показаниями индикатора.

Целевое утверждение программы:

project is ('MODEL')

Все подцели логической программы, вызываемые с префиксом `@`, автоматически объявляются акторами, при этом все аргументы акторных вызовов автоматически объявляются общими переменными. В Акторном Прологе все логические акторы используют свои собственные (*локальные*) значения общих переменных, а сопоставление этих значений происходит лишь в момент (успешного) завершения доказательства актора, а также в случае использования встроенных предикатов и операторов *':='*, *'in'*, *'copy'*. Противоречия, обнаруженные между локальными значениями акторов, устраняются путём повторного доказательства акторов, доказанных ранее, в соответствии с логической акторной моделью вычи-

```
class 'CONTROL' using 'TIMER' is
state
panel
entry
[
run:-                          -- Определение начального состояния
    @ initialize_state,        -- устройства.
    start.                     -- Запуск виртуального таймера.
--
initialize_state:-
    state == 0.0.              -- Встроенный предикат унификации
initialize_state:-             -- обозначается '=='.
    switch_state.
--
tick:-                         -- Обработка сигнала от таймера.
    switch_state.
--
switch_state:-
    panel ? get(entry,V),      -- Ввод значения V текстового поля entry.
    state := V.                -- Разрушающее присваивание.
]
```

Рис. 2.8: Определение класса 'CONTROL'.

```
class 'MODEL' using 'DIALOG' is
name     = "ВИРТУАЛЬНЫЙ ПРИБОР"
device3  = ('INDICATOR', panel= self, entry= 3, state= output)
output
[
goal:-
     shift,                  -- Разворачивание окна диалога.
     ('CONTROL',             -- Динамическое построение
          panel= self,       -- экземпляра класса.
          entry= 1,
          state= X) ? run,   -- Дальний вызов предиката.
     ('CONTROL',             -- Динамическое построение
          panel= self,       -- экземпляра класса.
          entry= 2,
          state= Y) ? run,   -- Дальний вызов предиката.
     @ sum(X,Y).             -- Построение актора.
--
sum(X,Y):-
     copy(X,Y),              -- Приём информации из других акторов.
     X == 0.0,               -- Встроенный предикат унификации
     output == Y,!.          -- обозначается '=='.
sum(X,Y):-
     copy(X,Y),              -- Приём информации из других акторов.
     Y == 0.0,
     output == X,!.
sum(X,Y):-
     copy(X,Y),              -- Приём информации из других акторов.
     R is X + Y,             -- Суммирование чисел.
     output := R.            -- Разрушающее присваивание.
--
action(5001):-               -- Обработка нажатия экранной кнопки 5001.
     break.                  -- Остановка программы (вызов
                             -- исключительной ситуации 0).
]
```

Рис. 2.9: Определение класса 'MODEL'.

слений.

2.5 Разработка Акторного Пролога

В приложении **A** приведено полное определение Акторного Пролога [22] от 22 марта 1998 г.

Разработанный логический язык отвечает требованиям, предъявляемым к современным языкам программирования, в том числе, требованиям к средствам раздельной трансляции и обработки исключительных ситуаций. Основные усилия в процессе разработки языка были направлены на получение по возможности более чёткого определения синтаксиса и семантики, допускающего простую и эффективную реализацию.

Одновременно с разработкой языка был создан прототип его интерпретатора, а также прототип транслятора Акторного Пролога в исходный текст промышленной системы логического программирования PDC PROLOG [10, 12, 77]. Эксперименты с прототипом языка показали возможность его эффективной реализации на персональных ЭВМ.

В настоящее время существуют интерпретатор языка, браузер — оболочка для визуального программирования на Акторном Прологе, а также транслятор в исходный текст на языке PDC PROLOG. Транслятор Акторного Пролога обеспечивает примерно 35% быстродействия исполняемого кода (в режиме «генерации типов» — до 85%) по сравнению с программами, написанными на языке PDC PROLOG вручную.

2.6 Выводы

Предложенный *метод интерактивного функционального моделирования* основан на использовании специального объектно-ориентированного логического языка — Акторного Пролога — для описания блоков и связей функциональных диаграмм в терминах логического программирования, а также осуществления логического анализа функциональных диаграмм, выявления и устранения смысловых противоречий, возникающих в процессе интерактивного изменения диаграммы человеком.

В соответствии с целями моделирования следует выделить три разновидности логических программ — *семантических моделей*: *структурные семантические модели*, *интерактивные семантические модели*, *имитационные семантические модели*, предполагающие использование

различных выразительных возможностей и механизмов логического языка.

Акторный Пролог реализует обобщённую *логическую акторную модель ФД информационной системы, проектируемой в интерактивном режиме*, представляющую информационную систему в виде некоторой теоремы на логическом языке, разделённой на *логические акторы — повторно доказываемые* подцели, взаимодействующие через *общие переменные*. Доказательство этой теоремы осуществляется в некотором *объектно-ориентированном пространстве поиска*, состоящем из отдельных миров, топология которого соответствует структуре ФД моделируемой системы.

Процесс человеко-машинного взаимодействия в логической акторной модели рассматривается как доказательство некоторой теоремы, в котором одновременно принимают участие человек, изменяющий условия задачи, и машина, обеспечивающая корректность доказательства. Предложен и разработан *механизм повторного доказательства* отдельных акторов, который обеспечивает корректность доказательства теоремы при изменении значений общих переменных и произвольном порядке доказательства акторов.

В целях реализации необходимых для интерактивного функционального моделирования средств, в Акторном Прологе разработана логическая интерпретация понятий ООП, отражающая *структурные*, *динамические* и *информационные* аспекты ООП.

Эксперименты с реализацией языка показали возможность его использования не только для интерактивного функционального моделирования, но также для имитационного моделирования и программирования информационных систем. Разработана *парадигма программирования*, соответствующая логической акторной модели вычислений.

Показана возможность эффективной реализации Акторного Пролога на персональных ЭВМ. В настоящее время существуют интерпретатор языка, браузер, а также транслятор в исходный текст на языке PDC PROLOG.

Новые версии определения Акторного Пролога регулярно публикуются на нашем WWW-сервере

http: // www.cplire.ru / Lab144 / index.html
(сервер ИРЭ РАН http: // www.cplire.ru /, раздел research divisions, biomedical diagnostic systems)

Глава 3

Декларативная и операционная семантика Акторного Пролога

В основе Акторного Пролога лежит *метод повторного доказательства подцелей*, идея которого заключается в следующем:

1. В логический язык вводятся специальные обозначения, с помощью которых в программе выделяются некоторые подцели (*логические акторы*), связанные общими переменными.

2. Исполнение логических программ осуществляется под управлением специальной стратегии (называемой в Акторном Прологе *механизмом повторного доказательства* или *акторным механизмом*), предусматривающей:

   (a) Возможность *изменения значений некоторых общих переменных* и последующего *повторного доказательства зависящих от них акторов* (если это понадобится для восстановления корректности и полноты логического вывода).

   (b) Возможность *параллельного исполнения повторных доказательств акторов*, когда это не нарушает корректность и полноту логического вывода.

   (c) Возможность *задержки восстановления состояний некоторых акторов при откате программы*, когда это не нарушает корректность и полноту логического вывода.

Параллельное исполнение и задержка восстановления акторов являются факультативными возможностями, поэтому, в целях упрощения изложения, формальное определение операционной семантики Акторного Пролога будет построено на основе некоторой *последовательной абстрактной машины логического вывода*. В целях упрощения изложения мы не будем также рассматривать вопросы, связанные с реализацией встроенных предикатов и средств управления Акторного Пролога.

3.1 Корректность логической интерпретации ООП в Акторном Прологе

Входным языком разработанной абстрактной машины логического вывода является хорновское подмножество [39, 14, 36] формул логики предикатов первого порядка, обогащённого синтаксическими средствами для реализации акторов. Все остальные объектно-ориентированные средства Акторного Пролога, а именно, средства, отражающие структурный и информационный аспекты ООП, однозначно и эффективно реализованы с помощью заданного входного языка.

Корректность такой реализации гарантируют следующие теоремы:

> **Теорема 3.1.1.** Программа на Акторном Прологе без средств управления, в которой используются «недоопределённые множества», может быть эффективно преобразована (с сохранением операционной семантики) в программу, не содержащую «недоопределённые множества».

Доказательство (конструктивное). Для того чтобы осуществить такое преобразование, достаточно:

1. Преобразовать все предложения, имитирующие логику 2-го порядка, в предложения с предикатным символом ’’, используя правило, заданное в разделе 6.1 определения Акторного Пролога (см. приложение **A** диссертации).

2. Представить все недоопределённые множества в программе в виде термов логики предикатов первого порядка с помощью глобальных преобразований, заданных в разделе 3.2.3 определения языка. При этом, если в состав рассматриваемого недоопределённого множества входит хвост R, необходимо осуществить следующие дополнительные действия:

- Если рассматриваемое недоопределённое множество входит в состав проекта или определения атрибутов некоторого класса, все вхождения переменной R в проекте или определении атрибутов должны быть связаны термом G_2, ограничивающим значение хвоста R (см. раздел 3.2.3 определения языка).
- Если недоопределённое множество входит в заголовок некоторого собственного предложения, в состав этого предложения непосредственно после заголовка, а также после каждой связки «или» (если они есть) должен быть добавлен вызов предиката унификации, связывающий переменную R с термом G_2.
- Если недоопределённое множество входит в подцель некоторого собственного предложения, в состав этого предложения непосредственно перед названной подцелью должен быть добавлен вызов предиката унификации, связывающий переменную R с термом G_2.

Использованные преобразования служат для определения операционной семантики рассматриваемых синтаксических конструкций, поэтому полученная программа, по построению, сохраняет операционную семантику исходной программы. ■

Теорема 3.1.2. Программа на Акторном Прологе, в которой используются «предикаты с переменным числом аргументов», может быть эффективно преобразована (с сохранением операционной семантики) в программу, не содержащую «предикаты с переменным числом аргументов».

Доказательство (конструктивное). Для того чтобы осуществить такое преобразование, достаточно:

1. Заменить все атомарные формулы вида $p(A_1, \ldots, A_n)$ в программе предикатами $p([A_1, \ldots, A_n])$.
2. Заменить в программе все предикаты с переменным числом аргументов — атомарные формулы вида $p(A_1, \ldots, A_n*)$ — предикатами $p([A_1, \ldots, A_{n-1}|A_n])$.

Использованные преобразования соответствуют определению операционной семантики предикатов с переменным числом аргументов, поэтому полученная программа, по построению, сохраняет операционную семантику исходной программы. ■

Теорема 3.1.3. Вещественные числовые литералы в программе на Акторном Прологе могут быть (эффективно) представлены в виде термов логики предикатов первого порядка.

Доказательство (конструктивное). Согласно определению вещественных числовых литералов Акторного Пролога (раздел 2.1.3 определения языка), для них задана максимальная относительная погрешность, определяемая в конкретных реализациях языка. Таким образом, любой вещественный литерал может быть представлен в виде пары целых чисел $\langle M, E \rangle$, где M — мантисса, а E — порядок вещественного числа. ■

Теорема 3.1.4. Программа на Акторном Прологе, в которой используются «объявления функций» и «вызовы функций», может быть эффективно преобразована (с сохранением операционной семантики) в программу, не содержащую «объявления и вызовы функций».

Доказательство (конструктивное). Для того чтобы осуществить такое преобразование, достаточно:

1. Преобразовать в предикаты все объявления функций в программе согласно правилам, заданным в разделе 6.3 определения языка.

2. Преобразовать в предикаты все вызовы функций согласно правилам, заданным в разделе 5.1.2 определения языка.

Использованные преобразования служат для определения операционной семантики рассматриваемых синтаксических конструкций, поэтому полученная программа, по построению, сохраняет операционную семантику исходной программы. ■

Теорема 3.1.5. Программа на Акторном Прологе без средств управления и акторных префиксов может быть эффективно преобразована (с сохранением операционной семантики) в программу на чистом Прологе (в формулу хорновского подмножества логики предикатов первого порядка).

Доказательство (конструктивное). Рассмотрим иерархию классов некоторой произвольной программы на Акторном Прологе без средств управления и акторных префиксов (рис. 3.1).

При условии отсутствия в программе акторов операционная семантика встроенного предиката *':='* (см. раздел 7.4 определения Акторного

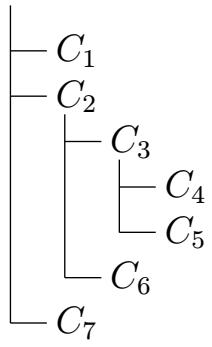


Рис. 3.1: Произвольная иерархия классов.

Пролога) в точности соответствует операционной семантике встроенного предиката унификации, встроенный предикат *in* (раздел 7.5 определения языка) вообще не влияет на исполнение программы, а операционная семантика остальных встроенных предикатов Акторного Пролога (раздел 8 определения языка) — предиката унификации *'=='* и предиката $fail$ — в точности соответствует операционной семантике этих предикатов в чистом Прологе («равно» и «ложь» соответственно).

Заменим в программе все вызовы встроенных предикатов *':='* и *'=='* равенствами, а вызовы встроенного предиката $fail$ — константой «ложь». Вызовы встроенного предиката *in* удалим из программы.

С учётом теорем 3.1.1, 3.1.2, 3.1.3, 3.1.4, предложения каждого отдельного класса C_i иерархии классов 3.1 можно представить в виде списка хорновских предложений S_{ir} (r=1,...,q):

$$S_{ir}: \quad p_{ir}(A_1, \ldots, A_n) :- Subgoals. \ ,$$

где $A_1, \ldots, A_n$ — параметры предиката p_{ir} в исходной программе, а $Subgoals$ — подцели предложения[1] S_{ir}. Дальние вызовы предикатов в составе подцелей $Subgoals$ мы, пока что, будем рассматривать просто как некоторые специальные атомарные формулы, а имена слотов в составе предложений — как некоторые специальные константы.

Для того чтобы преобразовать такую иерархию классов в список хорновских предложений, осуществим следующие действия:

1. Пронумеруем все классы C_i целыми числами $j = 1, \ldots, k$, обходя

[1]Согласно определению Акторного Пролога, в предложениях классов допускается использование связки «или», однако мы будем считать, что все такие предложения представлены в виде наборов хорновских предложений с одинаковыми заголовками.

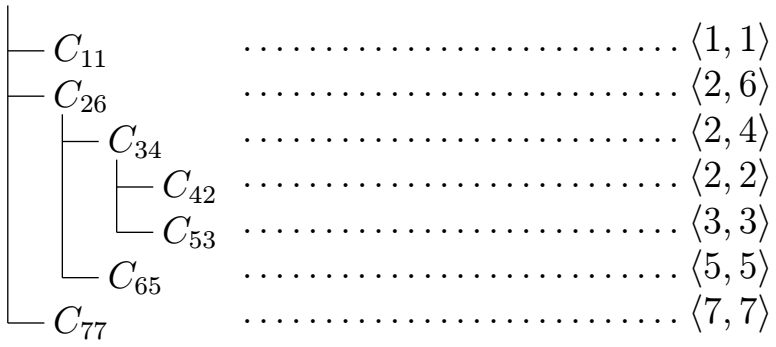


Рис. 3.2: Размеченное дерево классов.

дерево иерархии сверху-вниз, соблюдая при этом следующее правило: второй индекс j присваивается вершине C_i только после того, как мы пронумеруем таким образом всех потомков C_i в иерархии наследования.

Пример. Вершина C_3 на рис. 3.2 получит индекс $j = 4$, после того как её потомки C_4 и C_5 получат индексы $j = 2$ и $j = 3$ соответственно.

2. Поставим в соответствие каждой вершине C_{ij} пару чисел — «диапазон» — $\langle m, j\rangle$, где j — второй индекс вершины C_{ij}, а m — наименьшее значение среди вторых индексов всех потомков C_{ij} в иерархии наследования или, если C_{ij} не имеет потомков, значение её второго индекса j.

   Пример. Вершина C_{34} получит диапазон $\langle 2, 4\rangle$, а вершина C_{53} — $\langle 3, 3\rangle$ (см. рис. 3.2).

3. Теперь *экземпляр* любого класса C_{ij} можно однозначно описать с помощью двух термов логики предикатов первого порядка:

   (a) «ключа» j класса C_{ij};

   (b) списка слотов экземпляра класса.

   Это значит, что все инициализаторы и конструкторы (в том числе — проект программы — см. раздел 4.2 определения языка) можно представить в виде предикатов, вычисляющих термы логики первого порядка. При этом предикаты, моделирующие конструкторы, согласно правилам 4.1.4 определения языка, должны вычислять

«ключи» соответствующих миров и списки слотов, а также исполнять дальние вызовы предикатов *goal* в сформированных экземплярах классов.

Поставим в соответствие каждому предложению S_{ir} некоторое предложение S'_{ir}, добавив в заголовок и подцели предложения S_{ir} дополнительный аргумент $Slots$ (список пар «имя_слота: значение_слота»), имитирующий слоты классов. Все имена слотов в составе предложений S_{ir} заменим некоторыми уникальными переменными для хранения значений слотов. Кроме того, в списки подцелей предложений следует внести вызовы предикатов, имитирующие инициализаторы и конструкторы, а также вызовы специального предиката $is\_element(Name, Value, Slots)$, унифицирующего значение $Value$ каждого слота $Name$ с соответствующим компонентом списка $Slots$:

$$S'_{ir}: \quad p_{ir}(Slots, A_1, \ldots, A_n) :- Subgoals'.$$

4. Теперь поставим в соответствие каждому предложению S'_{ir} предложение S''_{ir}, добавив в заголовок предложения S'_{ir} ещё один дополнительный аргумент — некоторую уникальную переменную Key, а в начало предложения S'_{ir} — две специальные подцели $Key \geq m$, $Key \leq j$, где m и j — параметры диапазона $\langle m, j \rangle$ класса C_{ij}, которому принадлежит предложение S'_{ir}. Кроме того, нам понадобится добавить соответствующие дополнительные аргументы во все подцели предложений программы, обозначающие вызовы предикатов различных классов. Все дальние вызовы предикатов в составе предложений следует заменить ближними вызовами, задав в качестве значений дополнительных аргументов этих вызовов «ключи» и списки слотов, описывающие соответствующие миры:

$$S''_{ir}: \quad p_{ir}(Key, Slots, A_1, \ldots, A_n) :- Key \geq m, Key \leq j, Subgoals''.$$

Пример. Дальний вызов предиката в *экземпляре* класса C_{34} вида

$$target\_world \; ? \; p(A_1, \ldots, A_n)$$

будет смоделирован с помощью ближнего вызова

$$p(\mathbf{4}, Slots, A_1, \ldots, A_n),$$

а для того чтобы смоделировать дальний вызов предиката в экземпляре класса C_{53}, необходимо использовать «ключ» $Key = 3$:

$$p(\mathbf{3}, Slots, A_1, \ldots, A_n).$$

5. Построим список предложений S''_{ir} всех классов C_{ij}. Предложения S''_{ir}, соответствующие различным классам иерархии наследования, должны быть взаимно упорядочены в списке таким образом, чтобы предложения классов-потомков всегда предшествовали предложениям классов-предков.

Объединение списка предложений S''_{ir} с определениями вспомогательных предикатов, описывающих инициализаторы, конструкторы и проект программы, а также с определением вспомогательного предиката *is_element* является представлением исходной программы на Акторном Прологе в виде программы на чистом Прологе. Использованные преобразования осуществлялись в соответствии с определением операционной семантики классов, миров и наследования, поэтому полученная программа, по построению, сохраняет операционную семантику исходной программы на Акторном Прологе.

Конъюнкция всех перечисленных предложений является представлением исходной программы в виде формулы хорновского подмножества логики предикатов первого порядка. ■

Заметим, что декларативная семантика встроенного предиката *':='* равна декларативной семантике встроенного предиката унификации, встроенный предикат *in* не влияет на декларативную семантику программ, а декларативная семантика предиката унификации *'=='* и предиката $fail$ соответствует декларативной семантике этих предикатов в чистом Прологе — «равно» и «ложь».

Определение 3.1.6. *Декларативной семантикой* программы на Акторном Прологе без средств управления является декларативная семантика программы на чистом Прологе, соответствующей по теореме 3.1.5 исходной программе без акторных префиксов.

Доказанные теоремы гарантируют корректность логической интерпретации классов, миров, наследования, а также недоопределённых множеств и вещественных чисел, то есть средств, отражающих структурный и информационный аспекты ООП в логическом языке. Корректность интерпретации динамического аспекта ООП в Акторном Прологе гарантируют следующие теоремы:

Теорема 3.1.7 (о корректности). Механизм повторного доказательства Акторного Пролога с проверкой вхождений, без средств управления является корректной стратегией управления.

Доказательство. В соответствии с теоремой о корректности SLD-резолюции при любом выбранном *правиле вычислений*[2] ([36], С. 288), стандартная стратегия управления, лежащая в основе акторного механизма, расширенная возможностями повторного и параллельного исполнения подцелей (см. раздел 7.3 определения Акторного Пролога), является корректной стратегией управления, и созданное ей дерево доказательства (И-дерево) является корректным результатом обработки заданного целевого утверждения программы. Поэтому для того чтобы показать корректность акторного механизма в целом, необходимо убедиться, что поддерживаемые им возможности нейтрализации и задержки восстановления состояний акторов при откате *не могут* привести к пропуску или удалению каких-либо подцелей, входящих в состав дерева доказательства, создаваемого некоторой корректной стратегией управления. Опасность такой потери подцелей дерева доказательства возникает в следующих случаях:

1. Нейтрализация актора является ни чем иным как удалением соответствующей подцели из дерева доказательства. Однако, в соответствии с правилами 7.3.1 определения языка, для продолжения логического вывода требуется, чтобы все нейтрализованные акторы были доказаны повторно, и, следовательно, все удалённые подцели снова добавляются в дерево доказательства.

2. Задержка восстановления состояния некоторого актора при откате может привести к потере соответствующей ему подцели дерева доказательства, если в результате задержки этот актор навсегда окажется в состоянии «нейтральный». Однако, в соответствии с правилами 7.3.3 определения языка, акторы могут задерживаться лишь в состоянии «доказан», поэтому задержка восстановления состояния актора не может привести к потере соответствующей ему подцели дерева доказательства.

3. Задержка восстановления актора, находящегося в состоянии «доказан», может привести к потере подцелей дерева доказательства, соответствующих вложенным акторам, созданным в ходе построения

[2]Правилом вычислений называется правило, задающее порядок исполнения подцелей предложений.

указанного доказательства актора, если вложенные по отношению к нему акторы окажутся удалёнными в ходе отката. В соответствии с правилами 7.3.3 определения языка, акторы не разрешается задерживать в состояниях, которым соответствуют доказательства, в ходе которых были созданы новые акторы, поэтому рассмотренный случай потери подцелей дерева доказательства также исключён.

Кроме того, корректность результатов логического вывода могло бы нарушить восстановление состояния задержанного актора, произошедшее уже после завершения работы программы, так как оно вывело бы акторы программы из согласованного состояния. Чтобы предотвратить эту опасность, правила 7.3.3 определения языка разрешают восстановление состояний задержанных акторов лишь в том случае, если в программе существует хотя бы один активный актор, ещё не осуществивший согласование акторов. Таким образом, если значения общих переменных актора, задержанного в состоянии «доказан», вступят в противоречие с результатами доказательства других акторов, такая ситуация обязательно будет выявлена и устранена в ходе очередного согласования акторов программы (раздел 7.3 определения языка).

Вывод: дерево доказательства, построенное акторным механизмом, включает *все* подцели дерева доказательства, построенного корректной стратегией управления. Следовательно, акторный механизм и сам также является корректной стратегией управления. ■

Для того чтобы продемонстрировать важность доказанной теоремы, рассмотрим следующий пример.

Пример 3.1.8. Программа, осуществляющая взаимодействие с пользователем через некоторое диалоговое окно, выводит полученные результаты, заполняя различные поля в этом окне. Операция заполнения полей диалогового окна является примером *необратимого процесса*, поскольку в случае отката программы содержимое полей окна должно сохраняться, пока программа не занесёт в них новые значения. В «обычном» Прологе отсутствует какая-либо логически корректная интерпретация необратимых процессов во «внешнем мире», и поэтому даже наличие у логической программы строгой декларативной семантики не даёт совершенно никаких гарантий, что вся совокупность значений, выведенных на экран в процессе поиска решения, составит некоторый осмысленный, корректный ответ на запрос пользователя. В реальной практике программирования решение этой проблемы целиком возлагается на плечи программиста, которому вменяется в обязанность учесть все возможные пути исполнения

программы и вовремя обновлять значения полей диалоговых окон.

В Акторном Прологе простые случаи необратимых процессов, подобные рассмотренному, получают адекватную логическую интерпретацию на основе идеи *задержки восстановления состояний акторов при откате*. В частности, поля диалоговых окон, с точки зрения Акторного Пролога, являются переменными, значения которых обновляются путём повторного доказательства акторов. При этом *теорема о корректности акторного механизма* гарантирует, что, в случае успешного завершения, программа на Акторном Прологе, правильно описывающая предметную область, заполнит поля диалоговых окон некоторыми правильными, согласованными друг с другом значениями.

Таким образом, доказанная теорема обеспечивает возможность построения *логической интерпретации интерфейса «человек-машина»*, а также разработки соответствующих *средств поддержки интерактивного режима*, гарантирующих корректность результатов, получаемых в ходе совместной деятельности человека и машины.

Не менее интересным является также вопрос, как механизм повторного доказательства влияет на *полноту* логических программ относительно их декларативной семантики.

Для того чтобы проиллюстрировать свойства акторного механизма, связанные с проблемой полноты, покажем, что работу акторного механизма без задержки восстановления акторов, при отсутствии вложенных акторов можно смоделировать, обходя в различном порядке вершины полного дерева поиска[3] T_0 программы на «обычном» Прологе, соответствующей по теореме 3.1.5 программе на Акторном Прологе после удаления акторных префиксов.

Стандартная стратегия управления, положенная в основу акторного механизма, последовательно обходит вершины дерева T_0, как интерпретатор чистого Пролога, но при этом реализует следующую дополнительную возможность: при достижении тупиковой вершины X (см. рис. 3.3) на ИЛИ-дереве (интерпретатор «обычного» Пролога, достигнув этой точки, осуществил бы откат программы), Акторный Пролог делает попытку достичь успеха, нейтрализовав и доказав повторно некоторые доказанные ранее акторы NA. Поскольку повторно доказываемые акторы соответствуют вершинам, лежащим на уже пройденном отрез-

[3]Полным деревом поиска (пространством поиска, ИЛИ-деревом) называется подмножество полного дерева вычислений логической программы, ограниченное некоторым конкретным *правилом вычислений*.

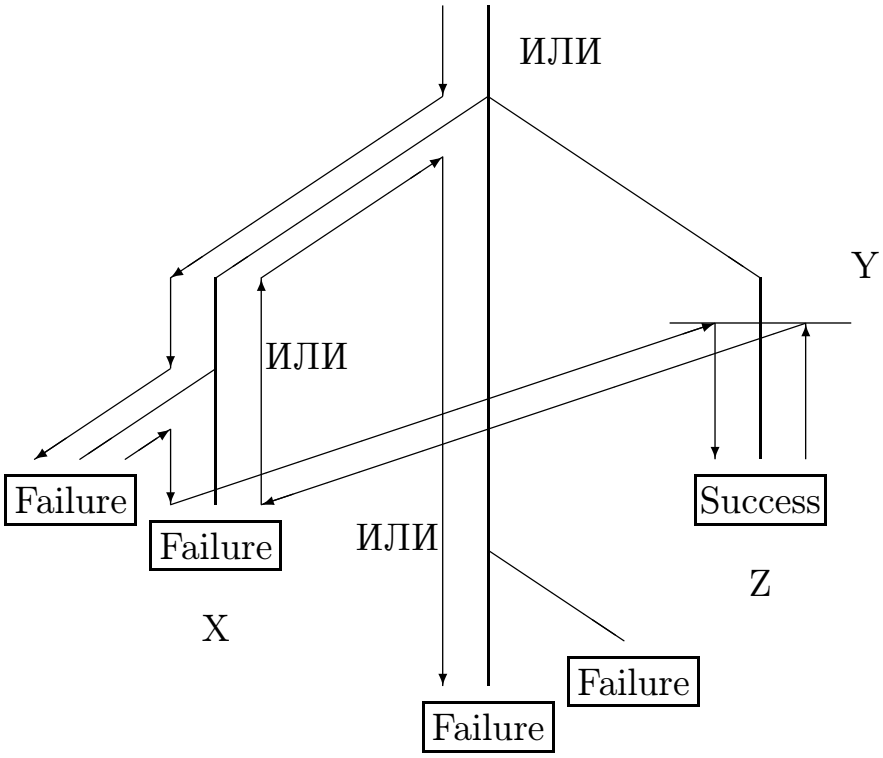


Рис. 3.3: Работа акторного механизма.

ке исследуемой ветви, такое «переключение» акторов NA можно смоделировать, осуществив переход из вершины X в другую вершину Y, на другой ветви дерева T_0 и попытавшись достичь успеха Z, продвигаясь из вершины Y. В случае если произойдёт откат программы, акторный механизм возвратится обратно в точку X, после чего продолжит использовать стандартную стратегию управления — произойдёт откат из вершины X и дальнейший последовательный обход дерева T_0 (в том числе, возможно, повторное рассмотрение вершины Y).

Ясно, что акторный механизм *не является справедливой стратегией управления*, поскольку в Акторном Прологе используется поиск в глубину, как в «обычном» Прологе, и программы на Прологе могут зацикливаться. Тем не менее, для того чтобы создавать программы, обеспечивающие полный перебор вариантов, необходимо быть уверенным, что пространство поиска программы на Акторном Прологе включает вычисления, выдающие *те же* ответы, которые нашла бы соответствующая (после удаления акторных префиксов) по теореме 3.1.5 программа, исполняемая под управлением некоторой корректной и полной стратегии.

Теорема 3.1.9 (о полноте). Если P_A — программа на Акторном Прологе без средств управления, а P_S — программа на чистом Прологе, соответствующая по теореме 3.1.5 программе P_A без акторных префиксов, то для любого успешного вычисления C_S в полном дереве поиска программы P_S, исполняемой под управлением стандартной стратегии, в полном дереве поиска программы P_A найдётся некоторое успешное вычисление C_A, выдающее тот же ответ, что и вычисление C_S.

Доказательство. Очевидно, что для вычисления искомых ответов интерпретатору Акторного Пролога нет необходимости использовать нейтрализацию и повторное доказательство акторов. Параллельное исполнение повторных доказательств акторов не может привести к потере ответов, так как правила раздела 7.3.2 определения языка устанавливают последовательный порядок доказательства (с перебором всех вариантов) тех акторов, между которыми выявлено взаимное влияние (выразившееся в нейтрализации и повторном доказательстве некоторых соответствующих им подцелей).

Поэтому для доказательства теоремы необходимо лишь показать, что возможности нейтрализации, повторного доказательства и задержки восстановления состояний акторов при откате

1. не могут помешать построению какого-либо успешного вычисления, а могут лишь привести к его преобразованию — изменению порядка выбора вызовов (возможно, сопровождаемого копированием вызовов) предикатов исполняемого целевого утверждения — с сохранением выдаваемого ответа.

2. не могут привести к бесконечным циклическим преобразованиям исследуемого пути вычисления.

Задержка восстановления состояний акторов при откате не может помешать построению какого-либо успешного вычисления, так как акторы (согласно правилам 7.3.3 определения языка) могут быть задержаны только в состоянии «доказан», и, следовательно, задержанные акторы обязательно будут нейтрализованы и доказаны повторно (раздел 7.3.1 определения языка), если их текущее состояние вступит в противоречие с новыми результатами логического вывода.

При этом, однако, задержка восстановления акторов может привести к тому, что в дальнейшем в ходе построения какого-либо другого вычисления C_A произойдут нейтрализация и повторное доказательство

задержанных акторов. Покажем, что такое событие не может помешать построению вычисления C_A.

Предположим, что в ходе построения вычисления C_A произошли нейтрализация и повторное доказательство некоторого актора. Поскольку повторно доказываемый актор соответствует некоторой уже пройденной вершине пути C_A, нейтрализация и повторное доказательство этого актора будет означать преобразование C_A — изменение порядка выбора вызовов предикатов исполняемого целевого утверждения. При этом, если в ходе повторного доказательства актора будут созданы новые (вложенные по отношению к нему) акторы, соответствующие подцелям, уже доказанным ранее, это будет соответствовать случаю копирования (дублирования) вызовов предикатов целевого утверждения.

В любом случае, построенное вычисление исполнит те же самые вызовы, что и вычисление C_S, которое было бы построено при отсутствии акторов, то есть оно выдаст тот же ответ, что и вычисление C_S.

Заметим, что задержка восстановления состояний акторов при откате никогда не может привести к задержке исполнения («повисанию») программы в ходе отката, так как, согласно правилам 7.3.3 определения языка, задержка восстановления акторов, возвращающихся в активное состояние, не допускается.

Бесконечные циклические преобразования пути C_A в ходе логического вывода также не возникают (нейтрализация акторов, сама по себе, никогда не может привести к зацикливанию логической программы) — это следует из того что:

1. Актор, вызвавший повторные доказательства других акторов, является активным, пока не завершатся все эти доказательства.

2. Акторный механизм (согласно правилам 7.3.1 определения языка) никогда не осуществляет нейтрализацию активных акторов.

То есть в ходе исследования пути C_A, соответствующего некоторому *конечному* (без рекурсивного построения бесконечного количества новых подцелей) вычислению C_S, процесс взаимной нейтрализации акторов может продолжаться лишь до тех пор, пока в программе остаются доказанные (*не* являющиеся активными) акторы.

Вывод: для любого произвольно выбранного успешного вычисления C_S полного дерева поиска программы P_S на чистом Прологе в полном дереве поиска программы P_A на Акторном Прологе без средств управле-

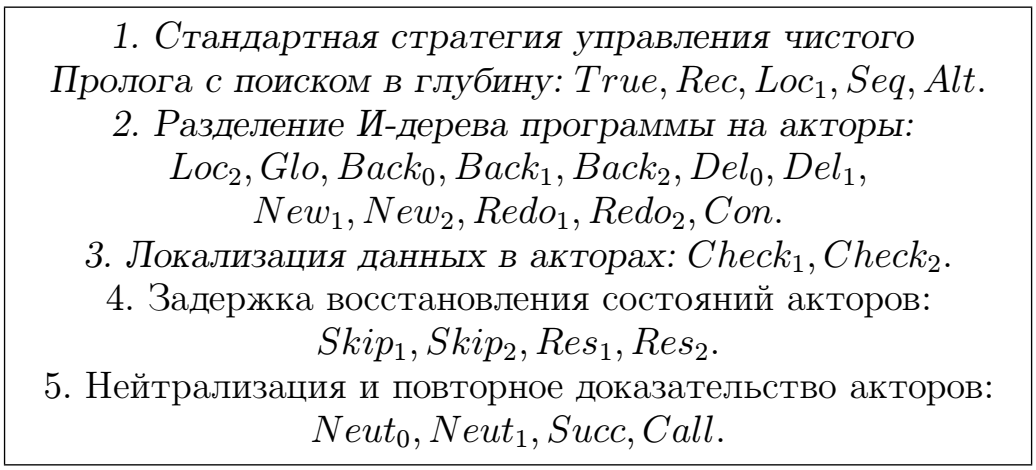


Рис. 3.4: Принцип работы акторной машины.

ния всегда найдётся успешное вычисление C_A, выдающее тот же ответ, что и вычисление C_S. ■

3.1.10. Следствие теоремы о полноте. Программа на Акторном Прологе без средств управления путём перебора вариантов обязательно найдёт *все* существующие решения задачи, если в ходе её исполнения не возникнут бесконечные вычисления.

Заметим, что теоремы о корректности и полноте акторного механизма 3.1.7, 3.1.9 умышленно рассматриваются независимо от определения операционной семантики акторной логической машины и соответствующих утверждений о свойствах этой машины, обсуждаемых в разделе 3.4. Разработанная абстрактная машина является лишь одним из возможных вариантов реализации стратегии управления Акторного Пролога, поэтому её следует рассматривать не как ещё одно определение акторного механизма, а, скорее, как некоторое промежуточное звено, помогающее корректно осуществить неформальный переход от определения Акторного Пролога к его программной реализации.

3.2 Архитектура акторной машины

Принцип работы акторной логической машины основан на пяти базовых идеях, условно изображённых на рис. 3.4 (вместе с соответствующими именами схем переходов, определение которых будет дано в разделе 3.3).

Стандартная стратегия управления чистого Пролога с поиском в глу-

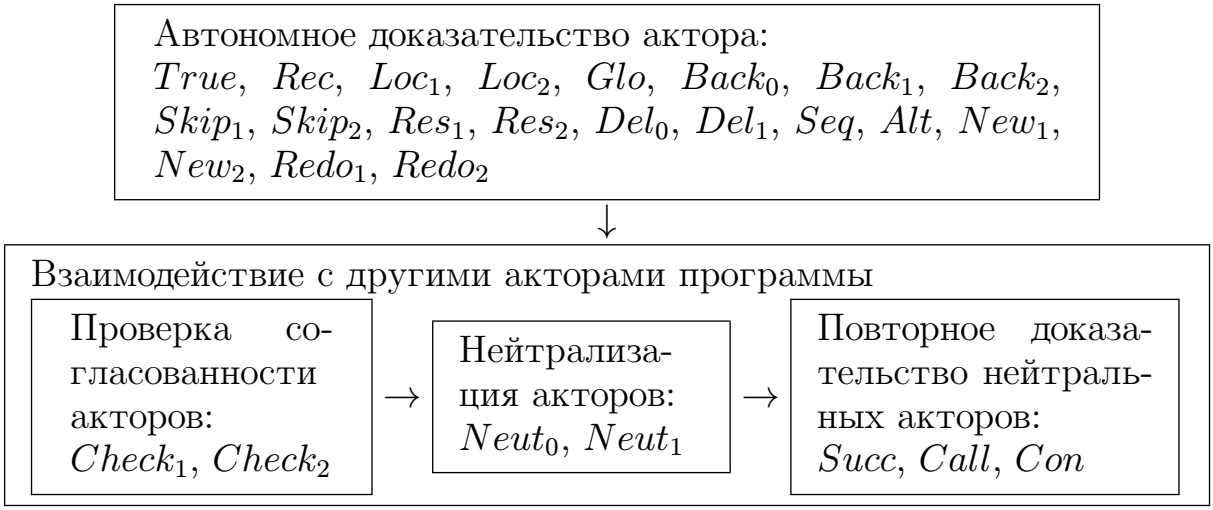


Рис. 3.5: Процесс доказательства логического актора.

бину является частным случаем стратегии управления Акторного Пролога. При этом логические акторы — до тех пор, пока не используются специальные возможности, предоставляемые акторной логической машиной — являются всего лишь «декоративной раскраской» И-дерева, не влияющей на стратегию управления.

Следующую базовую идею акторной машины можно условно назвать «локализацией данных в акторах». Каждый актор программы использует свои собственные («локальные») значения общих переменных, а сопоставление (проверка существования наиболее общего унификатора) локальных значений отдельных акторов осуществляется только в момент (успешного) завершения доказательства актора, а также при использовании некоторых встроенных средств языка.

В результате сопоставления значений переменных акторная машина может нейтрализовать некоторые акторы, чтобы тем самым обеспечить существование наиболее общего унификатора для всех оставшихся локальных значений в программе, и попытаться доказать их повторно (пятый слой на рис. 3.4).

Таким образом, процесс доказательства каждого отдельного актора программы включает основные фазы, перечисленные на рис. 3.5.

Согласно семантике акторной логической машины, актор может находиться в одном из трёх состояний[4] (см. рис. 3.6):

[4]В работах [26, 22] для обозначения соответствующих состояний акторов исполь-

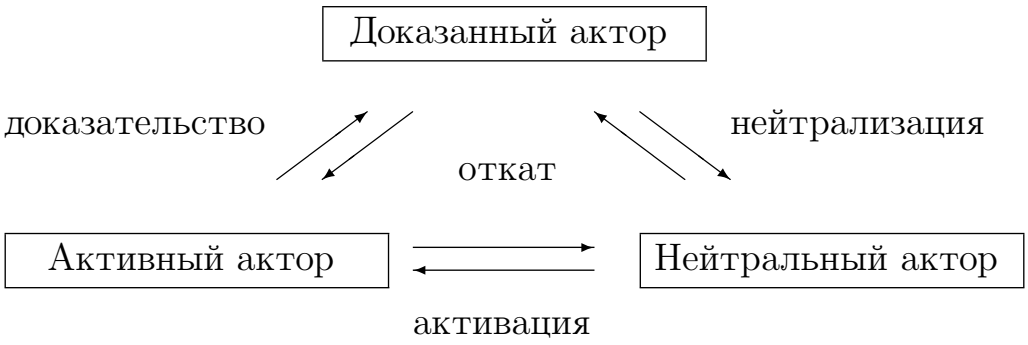


Рис. 3.6: Состояния логического актора.

1. Актор является «активным», если доказательство этой подцели ещё не окончено и продолжается в данный момент времени.

2. В том случае, если его доказательство благополучно завершилось успехом, актор становится «доказанным».

3. Актор, предыдущее доказательство которого отменено, а повторное ещё не началось, является «нейтральным».

Нейтрализация активных акторов не допускается — в разделе 3.3.2 будут определены строгие правила отбора акторов для нейтрализации — поэтому весьма возможно, что обнаруженные в процессе сопоставления локальных значений переменных противоречия не удастся устранить с помощью нейтрализации акторов — в таком случае используется стандартный механизм отката, и программа возвращается на этап автономного доказательства.

Если же процесс сопоставления проходит успешно, и в результате его оказывается нейтральным некоторое множество акторов NA, акторная машина пытается повторно построить доказательства всех нейтральных акторов. Если все (повторные) доказательства нейтральных акторов завершаются успехом (или если множество NA пустое), доказательство рассматриваемого актора считается успешно оконченным. В противном случае, если хотя бы один из нейтральных акторов не удаётся повторно доказать, происходит откат программы, возвращающий её на этап автономного доказательства.

зовались термины «активный», «неактивный» и «подавленный», однако они представляются не совсем удачными.

Чтобы определить процесс доказательства более формально, нам понадобятся следующие понятия:

- Состояние акторной машины — множество акторов, определяющее текущее состояние акторной машины:

$$\Gamma = \{A_1, A_2, \ldots, A_n\},$$

где A_i, $i = 1, \ldots, n$ — акторы программы.

- Актор — динамический объект, возникающий в результате «акторного» вызова предиката (процедуры программы):

$$A_i = \langle \alpha, m(t_1, \ldots, t_k), R \rangle,$$

где α — (уникальное) имя актора; $m(t_1, \ldots, t_k)$ — атомарная формула, соответствующая рассматриваемому актору; R — список «результаты доказательств актора».

- Результат доказательства актора — информация, полученная в ходе доказательства актора: связывания, точки выбора и т. п.

$$E = \langle \beta, F \rangle,$$

где β — имя актора, вызвавшего данное доказательство (доказательство может быть первым или повторным); F — «список продолжений при неудаче» — стек, с помощью которого моделируется процесс поиска с возвратом.

- Продолжение доказательства при неудаче — стек, с помощью которого моделируется доказательство последовательности подцелей без откатов:

$$C = \langle G, \sigma, N, B \rangle,$$

где G — «список подцелей»; σ — текущая подстановка, используемая в доказательстве списка подцелей; N — список имён акторов, нейтрализованных в процессе доказательства рассматриваемого списка подцелей; B — список имён акторов, созданных в процессе доказательства этих подцелей.

- Подцель — это вызов предиката (простой $m(t_1, \ldots, t_k)$ или «акторный» $@m(t_1, \ldots, t_k)$), композиция подцелей S_1 *and* S_2 и т. п.

Операционная семантика акторной логической машины определяется с помощью системы переходов между состояниями машины. Для описания системы переходов будет использован абстрактный @-язык с набором обозначений, заданным в таблицах 3.1, 3.2.

Формулы вида $\Gamma.\alpha\,\{GL = S : G, Subst = \sigma\}$ следует читать «в состоянии акторной машины Γ существует актор с именем α, обладающий следующими свойствами: значением ячейки "подцели" (GL) на вершине стека продолжений при неудаче, расположенного на вершине стека результатов доказательств актора, является список $S : G$, а значением ячейки "подстановка" (расположенной там же) — некоторая подстановка σ». Аналогично, формула $\Gamma.\alpha = \langle \alpha, M, R\rangle$ читается «в состоянии Γ существует актор с именем α, равный $\langle \alpha, M, R\rangle$». Приведённые формулы разрешается также использовать в значении «состояние Γ, в котором существует актор с именем α, обладающий следующими свойствами ...» При построении новых состояний акторной машины будут использоваться также формулы вида $\Gamma' = \Gamma : \alpha\,\{GL := G\}$ — «состояние Γ' отличается от Γ тем, что ячейке "подцели" на вершине стека продолжений при неудаче, расположенного на вершине стека результатов доказательств актора α, присвоено новое значение G».

Логическая программа, в терминах @-языка, задаётся определением процедур D и некоторым начальным состоянием

$$\Gamma^0(\tau) = \left\langle \tau, m(t_1, \ldots, t_k), \left\langle \xi, \langle m(t_1, \ldots, t_k) : nil, \varepsilon, [\,], [\,]\rangle : nil \right\rangle : nil \right\rangle,$$

где τ — имя некоторого «целевого» актора, активного в состоянии Γ, а ξ — условное имя «внешнего» (по отношению к машине) актора, осуществившего постановку задачи и ожидающего окончание доказательства. В состоянии Γ^0 все акторы логической машины, отличные от τ, должны быть «доказанными»[5]:

$$\forall\alpha : \Gamma^0.\alpha\,\{Name \neq \tau\} : is\_proven(\Gamma^0, \alpha)$$

Конечное состояние машины может быть либо *успешным*

$$\Gamma^{SUCCESS}.\tau\,\{Cont = \langle success, \sigma, N, B\rangle\},$$

либо *неуспешным*

$$\Gamma^{FAILURE}.\tau\,\{FL = \langle failure, \varepsilon, [\,], [\,]\rangle : nil\},$$

[5]Определение предиката *is_proven* будет дано в разделе 3.3.2.

Таблица 3.1: Основные обозначения @-языка.

Понятие	Общее обозначение	Определение и структура объектов	Представители
Предметная константа	$Const$		a, b, c
Предметная переменная	Var		X, Y, Z
Функциональный символ	Fun		f
Терм	$Term$	$Const$ Var $f(t_1, \ldots, t_k)$	t, v, u
Атом	$Atom$	$m(t_1, \ldots, t_k)$	M
Имя актора	$Name$	$\alpha, \beta, \gamma, \ldots; \tau; \xi; \phi$, где τ — имя «целевого» актора, ξ — имя «внешнего» актора, ϕ — используется в качестве фиктивного имени актора	
Подцель	$Subgoal$	$true$, $fail$, M, $@M$ S_1 *and* S_2, S_1 *or* S_2 $del([\alpha_1, \ldots, \alpha_n])$ $back([\alpha_1, \ldots, \alpha_n])$ $wait(\gamma)$, $redo(\gamma)$ $neutralize(\{\alpha_1, \ldots, \alpha_n\})$ $restart(\{\alpha_1, \ldots, \alpha_n\})$	S
Процедура	$Procedure$	$M :- S$	P
Определение процедур*	$Decl$	Функция $Atom \to Subgoal$	D

*В целях упрощения синтаксиса языка принимается соглашение о том, что в определении программы нельзя использовать процедуры с одинаковым функтором (именем и арностью) заголовка M.

Таблица 3.2: Определение акторной логической машины.

Понятие	Обозначение	Определение и структура объектов	Представители
Состояние акторной машины	$State$	$\{A_1, A_2, \ldots, A_n\}$	Γ
Актор	$Actor$	$\langle \alpha, M, R \rangle$	A
Результаты доказательств	RL	nil $E : R$, список, где E — голова, R — хвост	R
Результат доказательства	$Result$	$\langle \beta, F \rangle$ $neutral$ — специальная константа	E
Продолжения при неудаче	FL	nil $C : F$	F
Продолжение при неудаче	$Cont$	$\langle G, \sigma, N, B \rangle$	C
Подцели	GL	nil $success$ $failure$ $S : G$	G
Подстановка	$Subst$	$\sigma, \theta, \ldots; \varepsilon$, где ε — пустая подстановка	
Имена нейтрализованных акторов	$Neutr$	$[\alpha_1, \ldots, \alpha_n]$	N
Имена созданных акторов	$Built$	$[\alpha_1, \ldots, \alpha_n]$	B

где τ — целевой актор, заданный в начальном состоянии Γ^0. Заметим, что успешное и неуспешное конечные состояния, согласно приведённому определению, являются *взаимоисключающими.*

Дедлоков в акторной логической машине, к счастью, не бывает.

3.3 Система переходов

Система переходов акторной логической машины определяется набором схем переходов и множеством меток Λ (обозначение типичного представителя — l), необходимым для разметки конкретных применений схем переходов.

Рассмотрим различные этапы доказательства логического актора, перечисленные на рис. 3.5.

3.3.1 Автономное доказательство актора

На первом этапе доказательства актора исполнение программы осуществляется в полном соответствии со стандартной стратегией управления (стратегией поиска «слева направо в глубину с возвратом»). Основными «работающими» на этой стадии схемами являются $True$, Rec, Loc_1, Seq и Alt, моделирующие поиск с откатом. При этом, однако, используется большое число вспомогательных схем, обслуживающих создание, удаление и восстановление состояния акторов. Особое значение имеют схемы $Skip_1$, $Skip_2$, Res_1 и Res_2, необходимые для интерпретации в идеологии языка необратимых процессов.

$True$ — удаление подцели $true$.

$$\frac{\Gamma.\alpha\,\{GL = G\} \xrightarrow{l} \Gamma'}{\Gamma.\alpha\,\{GL = true : G\} \xrightarrow{l} \Gamma'}$$

или, словами, «если является допустимым переход $\Gamma.\alpha\,\{GL = G\} \xrightarrow{l} \Gamma'$, то существует также и переход $\Gamma.\alpha\,\{GL = true : G\} \xrightarrow{l} \Gamma'$».

Rec — вызов предиката. Функция $rename : P \to P'$ осуществляет переименование переменных процедуры, функция $mgu : (M_1, M_2) \to \sigma$ вычисляет наиболее общий унификатор выражений M_1, M_2 (если он су-

ществует).

$$\frac{\begin{array}{l}\Gamma.\alpha\,\{GL = m(t_1,\ldots,t_k) : G, Subst = \sigma\}\\ \exists\, P \in D :\\ \quad rename(P) = (m(u'_1,\ldots,u'_k) :\!- S')\ \wedge\\ \quad \exists\,\theta = mgu(m(t_1,\ldots,t_k)\sigma, m(u'_1,\ldots,u'_k))\end{array}}{\begin{array}{l}\Gamma' = \Gamma : \alpha\,\{GL := S' : G, Subst := \sigma\theta\}\\ \Gamma \xrightarrow{\langle Rec,\alpha\rangle} \Gamma'\end{array}}$$

где $\langle Rec, \alpha\rangle$ — метка рассматриваемой схемы переходов.

Loc_1 — «локальный» откат. Функция '$-$' обозначает разность списков: $L - L' = L''$, если $L'' = [\alpha_1,\ldots,\alpha_n]$ и $L = [\alpha_1,\ldots,\alpha_n|L']$. Функция '$+$' — конкатенация списков.

$$\frac{\begin{array}{l}\Gamma.\alpha\,\{FL = \langle S : G, \sigma, N, B\rangle : (\langle G', \sigma', N', B'\rangle : F')\},\\ S = fail\ \vee\\ (S = m(t_1,\ldots,t_k)\ \wedge\ \neg\exists\, P \in D :\\ \quad rename(P) = (M' :\!- S')\ \wedge\\ \quad \exists\,\theta = mgu(m(t_1,\ldots,t_k)\sigma, M'))\end{array}}{\begin{array}{l}\Gamma' = \Gamma : \alpha\,\{FL := \langle back(N'' + B'') : (del(B'') : G'), \sigma', N', B'\rangle : F'\},\\ N'' = N - N',\ B'' = B - B'\\ \Gamma \xrightarrow{\langle Loc_1,\alpha\rangle} \Gamma'\end{array}}$$

Loc_2 — обнаружение необходимости глобального отката.

$$\frac{\begin{array}{l}\Gamma.\alpha\,\{FL = \langle S : G, \sigma, N, B\rangle : \underline{nil}\},\\ S = fail\ \vee\\ (S = m(t_1,\ldots,t_k)\ \wedge\ \neg\exists\, P \in D :\\ \quad rename(P) = (M' :\!- S')\ \wedge\\ \quad \exists\,\theta = mgu(m(t_1,\ldots,t_k)\sigma, M'))\end{array}}{\begin{array}{l}\Gamma' = \Gamma : \alpha\,\{FL := \langle back(N + B) : (del(B) : failure), \varepsilon, [\,], [\,]\rangle : nil\}\\ \Gamma \xrightarrow{\langle Loc_2,\alpha\rangle} \Gamma'\end{array}}$$

Glo — «глобальный» откат.

$$\frac{\begin{array}{l}\Gamma.\alpha\,\{Result = \langle\beta, \langle failure, \varepsilon, [\,], [\,]\rangle : nil\rangle\}\\ \Gamma.\beta\,\{Subgoal = wait(\alpha)\}\end{array}}{\begin{array}{l}\Gamma' = \Gamma : \beta\,\{GL := fail : nil\}\\ \Gamma \xrightarrow{\langle Glo,\beta,\alpha\rangle} \Gamma'\end{array}}$$

$Back_0$ — завершение восстановления состояний акторов.

$$\frac{\Gamma.\alpha\,\{GL = G\} \xrightarrow{l} \Gamma'}{\Gamma.\alpha\,\{GL = back([\,]) : G\} \xrightarrow{l} \Gamma'}$$

$Back_1$ — восстановление актора (активного или доказанного).

$$\frac{\begin{array}{l}\Gamma.\alpha\,\{GL = back([\gamma|BList]) : G_\alpha\} \\ \Gamma.\gamma\,\{RL = \langle\beta, \langle G_\gamma, \sigma_\gamma, N_\gamma, B_\gamma\rangle : F_\gamma\rangle : R_\gamma\},\ \beta \neq \phi\end{array}}{\begin{array}{l}\Gamma' = \Gamma : \alpha\,\{GL := back(N_\gamma + B_\gamma + BList) : (del(B_\gamma) : G_\alpha)\}, \\ \qquad\quad \gamma\,\{RL := R_\gamma\} \\ \Gamma \xrightarrow{\langle Back_1,\alpha,\gamma\rangle} \Gamma'\end{array}}$$

$Back_2$ — восстановление нейтрального актора.

$$\frac{\begin{array}{l}\Gamma.\alpha\,\{GL = back([\gamma|BList]) : G_\alpha\} \\ \Gamma.\gamma\,\{RL = neutral : R_\gamma\}\end{array}}{\begin{array}{l}\Gamma' = \Gamma : \alpha\,\{GL := back(BList) : G_\alpha\}, \\ \qquad\quad \gamma\,\{RL := R_\gamma\} \\ \Gamma \xrightarrow{\langle Back_2,\alpha,\gamma\rangle} \Gamma'\end{array}}$$

$Skip_1$ — пропуск восстановления состояния актора.

$$\frac{\begin{array}{l}\Gamma.\alpha\,\{GL = back([\gamma|BList]) : G_\alpha\} \\ \Gamma.\gamma\,\{Result = \langle\beta, \langle G_\gamma, \sigma_\gamma, N_\gamma, [\,]\rangle : F_\gamma\rangle\},\ \beta \neq \phi\end{array}}{\begin{array}{l}\Gamma' = \Gamma : \alpha\,\{GL := back(N_\gamma + BList) : G_\alpha\}, \\ \qquad\quad \gamma\,\{Result := \langle\phi, \langle G_\gamma, \sigma_\gamma, [\,], [\,]\rangle : nil\rangle\} \\ \Gamma \xrightarrow{\langle Skip_1,\alpha,\gamma\rangle} \Gamma'\end{array}}$$

$Skip_2$ — сохранение текущего состояния актора.

$$\frac{\begin{array}{l}\Gamma.\alpha\,\{GL = back([\gamma|BList]) : G_\alpha\} \\ (\Gamma.\gamma\,\{RL = \langle\phi, F_\phi\rangle : (\langle\beta, \langle G_\gamma, \sigma_\gamma, N_\gamma, [\,]\rangle : F_\gamma\rangle : R_\gamma)\} \vee \\ \ \Gamma.\gamma\,\{RL = \langle\phi, F_\phi\rangle : (\langle\beta, neutral : F_\gamma\rangle : R_\gamma)\}, \\ \ \text{во втором случае будем считать } N_\gamma = [\,]), \\ \ \beta \neq \phi\end{array}}{\begin{array}{l}\Gamma' = \Gamma : \alpha\,\{GL := back(N_\gamma + BList) : G_\alpha\}, \\ \qquad\quad \gamma\,\{RL := \langle\phi, F_\phi\rangle : R_\gamma\} \\ \Gamma \xrightarrow{\langle Skip_2,\alpha,\gamma\rangle} \Gamma'\end{array}}$$

Res_1 — спонтанное (отложенное) восстановление состояния актора.

$$\frac{\Gamma.\gamma\,\{RL = R\} \xrightarrow{l} \Gamma'}{\Gamma.\gamma\,\{RL = \langle \phi, F\rangle : R\} \xrightarrow{l} \Gamma'}$$

Res_2 — удаление предыдущего состояния актора.

$$\frac{\Gamma.\gamma\,\{RL = \langle \phi, F\rangle : R'\} \xrightarrow{l} \Gamma'}{\Gamma.\gamma\,\{RL = \langle \phi, F\rangle : (\langle \phi, F'\rangle : R')\} \xrightarrow{l} \Gamma'}$$

Del_0 — завершение удаления акторов.

$$\frac{\Gamma.\alpha\,\{GL = G\} \xrightarrow{l} \Gamma'}{\Gamma.\alpha\,\{GL = del([\,]) : G\} \xrightarrow{l} \Gamma'}$$

Del_1 — удаление актора. Функция ‘/’ обозначает удаление актора: $\Gamma_1/\gamma = \Gamma_2$, такое что $\{\langle \gamma, M_\gamma, R_\gamma\rangle\} \cup \Gamma_2 = \Gamma_1$, $\Gamma_1 \neq \Gamma_2$.

$$\frac{\Gamma.\alpha\,\{Subgoal = del([\gamma|DList])\}}{\Gamma' = (\Gamma : \alpha\,\{Subgoal := del(DList)\})\,/\gamma}$$
$$\Gamma \xrightarrow{\langle Del_1, \alpha, \gamma\rangle} \Gamma'$$

Схемы $Skip_1$, $Skip_2$, Res_1, Res_2 (а также $Check_1$, $Neut_1$ и $Call$) придают акторной логической машине недетерминированность, которая является необходимым условием для построения логической интерпретации информационных систем, «открытых» по отношению к внешнему миру [37, 65, 21]. При этом $Skip_1$ и $Skip_2$ позволяют моделировать сохранение состояний некоторых акторов в случае отката программы, а Res_1 — их спонтанное восстановление.

Seq — исполнение конъюнкции подцелей.

$$\frac{\Gamma.\alpha\,\{GL = S_1 : (S_2 : G)\} \xrightarrow{l} \Gamma'}{\Gamma.\alpha\,\{GL = (S_1\ and\ S_2) : G\} \xrightarrow{l} \Gamma'}$$

Alt — исполнение дизъюнкции подцелей.

$$\frac{\Gamma.\alpha\,\{FL = \langle S_1 : G, \sigma, N, B\rangle : (\langle S_2 : G, \sigma, N, B\rangle : F)\} \xrightarrow{l} \Gamma'}{\Gamma.\alpha\,\{FL = \langle (S_1\ or\ S_2) : G, \sigma, N, B\rangle : F\} \xrightarrow{l} \Gamma'}$$

$code : [[t_1, \ldots, t_n], \sigma] \to [[t'_1, \ldots, t'_n], \sigma']$
$\sigma' := \sigma;$
$do\ \ i = 1, \ldots, n$
$\quad if\ \ (t_i = f(u_1, \ldots, u_k)$ — составной терм)
$\quad\quad [[v_1, \ldots, v_k], \sigma'] := code([u_1, \ldots, u_k], \sigma);$
$\quad\quad t'_i := f(v_1, \ldots, v_k);$
$\quad\quad \sigma := \sigma'$
$\quad elsif\ \ (t_i$ — переменная)
$\quad\quad if\ \ (t_i\sigma$ — переменная) $\ t'_i := t_i$
$\quad\quad else\ \ [t'_i, \sigma'] := copy(t_i, \sigma);\ \ \sigma := \sigma'$
$\quad\quad fi$
$\quad else\ \ t'_i := t_i$
$\quad fi$
od

Рис. 3.7: Определение функции кодирования.

New_1 — исполнение акторного вызова предиката. Для определения этой схемы понадобится ввести специальную функцию «кодирования» аргументов предиката.

Функция кодирования *code* (рис. 3.7) обеспечивает передачу в создаваемый актор максимального количества информации о значениях аргументов: если переменные, передаваемые в составе аргументов, хотя бы частично означены в акторе, осуществляющем вызов рассматриваемой акторной подцели, вместо них передаются константы и составные термы, а новыми переменными заменяются только неопределённые компоненты значений аргументов.

Вспомогательная функция *copy* (рис. 3.8) осуществляет копирование значений переменных, а вспомогательная функция *new_variable*() — построение новых переменных.

Рассмотренные преобразования осуществляются с целью:

1. Максимально увеличить количество информации, передаваемой в создаваемый актор перед началом его доказательства.

2. По возможности уменьшить количество создаваемых общих переменных (хотя в некоторых случаях использование функции *code* может вызвать прямо противоположный эффект).

$copy : [t, \sigma] \to [t', \sigma']$
if $(t\sigma = f(u_1, \ldots, u_k)$ — составной терм)
$\sigma' := \sigma;$
do $i = 1, \ldots, k$
if $(u_i\sigma$ — переменная)
$u'_i := new\_variable();$
$\sigma' := \sigma \cup \{u'_i = u_i\}$ — новая подстановка
$else$ $[u'_i, \sigma'] := copy(u_i, \sigma)$
$fi;$
$\sigma := \sigma'$
$od;$
$t' := f(u'_1, \ldots, u'_k)$
$else$ $t' := t\sigma;$ $\sigma' := \sigma$
fi

Рис. 3.8: Определение функции копирования.

Кроме того, необходимо отметить, что наличие такого достаточно сложного кодирования имеет принципиальное значение с точки зрения выразительной мощности языка, так как обеспечивает возможность строить бесконечное количество акторов с различной структурой (конкретно, отличающихся во втором компоненте кортежа $\langle \alpha, M, R \rangle$) во время исполнения программы конечной длины.

Выражение $not\_exists(\Gamma, \gamma)$ означает $\langle \gamma, M_\gamma, F_\gamma \rangle \notin \Gamma$.

$$\frac{\begin{array}{l}\Gamma.\alpha\,\{FL = \langle @m(t_1, \ldots, t_k) : G, \sigma, N, B\rangle : F\} \\ not\_exists(\Gamma, \gamma) \\ \exists\, P \in D : \\ \quad rename(P) = (m(u'_1, \ldots, u'_k) :- S') \ \wedge \\ \quad ([v_1, \ldots, v_k], \sigma') := code([t_1, \ldots, t_k], \sigma) \wedge \\ \quad \exists\, \theta = mgu(m(v_1, \ldots, v_k), m(u'_1, \ldots, u'_k))\end{array}}{\begin{array}{l}\Gamma' = \left(\Gamma : \alpha \left\{ \begin{array}{l} FL := \langle wait(\gamma) : G, \underline{\sigma'}, N, [\gamma|B]\rangle \\ : (\langle redo(\gamma) : G, \underline{\sigma'}, N, [\gamma|B]\rangle : F) \end{array} \right\}\right) \cup \\ \quad \{\langle \gamma, m(v_1, \ldots, v_k), \langle \alpha, \langle S' : nil, \underline{\theta}, [\,], [\,]\rangle : nil\rangle : nil\rangle\} \\ \Gamma \xrightarrow{\langle New_1, \alpha, \gamma\rangle} \Gamma'\end{array}}$$

New_2 — обнаружение невозможности акторного вызова.

$$\frac{\begin{array}{l}\Gamma.\alpha\,\{Subgoal = @m(t_1,\ldots,t_k), Subst = \sigma\}\\ \neg\exists\, P \in D:\\ \quad rename(P) = (m(u'_1,\ldots,u'_k) :- S')\ \wedge\\ \quad ([v_1,\ldots,v_k], \sigma') := code([t_1,\ldots,t_k], \sigma)\ \wedge\\ \quad \exists\,\theta = mgu(m(v_1,\ldots,v_k), m(u'_1,\ldots,u'_k))\end{array}}{\begin{array}{l}\Gamma' = \Gamma : \alpha\,\{GL := fail : nil\}\\ \Gamma \xrightarrow{\langle New_2,\alpha\rangle} \Gamma'\end{array}}$$

$Redo_1$ — откат акторного вызова метода.

$$\frac{\begin{array}{l}\Gamma.\alpha\,\{FL = \langle redo(\gamma) : G_\alpha, \sigma_\alpha, N_\alpha, B_\alpha\rangle : F_\alpha\}\\ \Big(\Gamma.\gamma\,\Big\{RL = \Big\langle\alpha, \langle G_\gamma, \sigma_\gamma, N_\gamma, B_\gamma\rangle : \Big(C'_\gamma : F'_\gamma\Big)\Big\rangle : R_\gamma\Big\}\ \vee\\ \ \Gamma.\gamma\,\Big\{RL = \langle\phi, F_\phi\rangle : \Big(\Big\langle\alpha, \langle G_\gamma, \sigma_\gamma, N_\gamma, B_\gamma\rangle : \Big(C'_\gamma : F'_\gamma\Big)\Big\rangle : R_\gamma\Big)\Big\}\Big)\end{array}}{\begin{array}{l}\Gamma' = \Gamma : \alpha\left\{\begin{array}{l} FL := \langle wait(\gamma) : G_\alpha, \sigma_\alpha, N_\alpha, B_\alpha\rangle\\ : (\langle redo(\gamma) : G_\alpha, \sigma_\alpha, N_\alpha, B_\alpha\rangle : F_\alpha)\end{array}\right\},\\ \qquad\qquad \gamma\,\Big\{RL := \Big\langle\alpha, \langle fail : nil, \sigma_\gamma, N_\gamma, B_\gamma\rangle : \Big(C'_\gamma : F'_\gamma\Big)\Big\rangle : R_\gamma\Big\}\\ \Gamma \xrightarrow{\langle Redo_1,\alpha,\gamma\rangle} \Gamma'\end{array}}$$

$Redo_2$ — обнаружение невозможности отката акторного вызова.

$$\frac{\begin{array}{l}\Gamma.\alpha\,\{Subgoal = redo(\gamma)\}\\ (\Gamma.\gamma\,\{RL = \langle\alpha, C_\gamma : \underline{nil}\rangle : R_\gamma\}\ \vee\\ \ \Gamma.\gamma\,\{RL = \langle\phi, F_\phi\rangle : (\langle\alpha, C_\gamma : \underline{nil}\rangle : R_\gamma)\})\end{array}}{\begin{array}{l}\Gamma' = \Gamma : \alpha\,\{GL := fail : nil\}\\ \Gamma \xrightarrow{\langle Redo_2,\alpha,\gamma\rangle} \Gamma'\end{array}}$$

3.3.2 Взаимодействие акторов

На втором этапе доказательства актора осуществляется сопоставление используемых им локальных значений переменных со значениями, используемыми в других акторах программы, а также нейтрализация и повторное доказательство акторов.

$Check_1$ — проверка согласованности акторов программы. Для определения этой схемы понадобится ввести некоторые дополнительные понятия и обозначения:

- $is\_neutral(\Gamma, \gamma)$ — предикат $\Gamma.\gamma\,\{Result = neutral\}$;

- $is\_active(\Gamma, \gamma) \stackrel{def}{=} \neg is\_neutral(\Gamma, \gamma) \ \wedge \ \Gamma.\gamma\,\{GL \neq success\}$;
- $is\_proven(\Gamma, \gamma) \stackrel{def}{=} \neg is\_neutral(\Gamma, \gamma) \ \wedge \ \Gamma.\gamma\,\{GL = success\}$;
- $SUBST(\Gamma, \gamma)$ — подстановка σ_γ актора γ, $\Gamma.\gamma\,\{Subst = \sigma_\gamma\}$, или пустая подстановка ε, если $is\_neutral(\Gamma, \gamma)$;
- $does\_exist(\Gamma, \gamma)$ означает $\langle \gamma, M_\gamma, R_\gamma \rangle \in \Gamma$;
- $\Sigma(\Gamma, \{\alpha_1, \ldots, \alpha_n\}) = \bigcup\limits_{i=1}^{n} SUBST(\Gamma, \alpha_i)$ — множество подстановочных равенств акторов $\alpha_1, \ldots, \alpha_n$ в состоянии Γ.

Определение 3.3.2.1. Множество S подстановочных равенств называется *противоречивым*, если существуют два таких его подмножества σ_1 и σ_2, являющиеся подстановками, что некоторая переменная X в результате применения этих подстановок получает значения $X\sigma_1$ и $X\sigma_2$, не имеющие наиболее общего унификатора:

$$inconsistent(S) \stackrel{def}{=} \exists\,\sigma_1 \subset S \ \wedge \ \exists\,\sigma_2 \subset S \ \wedge \ \exists\,X : \neg\exists\, mgu(X\sigma_1, X\sigma_2).$$

Определение 3.3.2.2. *Согласованное* множество подстановочных равенств: $consistent(S) \stackrel{def}{=} \neg inconsistent(S)$.

Определение 3.3.2.3. Множество имён акторов NA для нейтрализации и последующего повторного доказательства $may\_be\_neutralized(\Gamma, NA)$:

1. $\forall \beta \in NA : does\_exist(\Gamma, \beta) \ \wedge \ is\_proven(\Gamma, \beta)$;
2. $\forall \beta \in NA$:
   $\exists$ множество $\{\alpha_i\}, i = 1, \ldots, k : does\_exist(\Gamma, \alpha_i)$:
   $inconsistent(\Sigma(\{\alpha_1, \ldots, \alpha_k, \beta\})) \ \wedge \ consistent(\Sigma(\{\alpha_1, \ldots, \alpha_k\}))$;
3. Множество подстановочных равенств акторов любого подмножества Γ, не пересекающегося с множеством акторов NA, является согласованным.

Условие (2) призвано исключить неоправданную нейтрализацию акторов, не имеющих отношения к противоречиям, обнаруженным между локальными значениями общих переменных.

$$\frac{\begin{array}{l}\Gamma.\alpha\,\{GL = nil\} \\ \exists\, NA : may\_be\_neutralized(\Gamma, NA)\end{array}}{\begin{array}{l}\Gamma' = \Gamma : \alpha\,\{GL := neutralize(NA) : (restart(NA) : success)\} \\ \Gamma \xrightarrow{\langle Check_1, \alpha\rangle} \Gamma'\end{array}}$$

Обратите внимание, что предлагаемое определение множества акторов для повторного доказательства несколько отличается от приведённого в [26]. В частности, в новом определении не понадобилось понятие «корневой актор», так как все акторы рассматриваются на общих основаниях, и, кроме того, в [26] недостаточно чётко определено, какие акторы разрешается нейтрализовывать и доказывать повторно, а какие нет. Ещё одним отличием рассматриваемой операционной семантики от [26] является возможность построения точек выбора (точек возврата) в результате *повторных* доказательств (недетерминированных) акторов.

$Check_2$ — обнаружение невозможности согласовать акторы.

$$\frac{\begin{array}{l}\Gamma.\alpha\,\{GL = nil\} \\ \neg\exists\, NA : may\_be\_neutralized(\Gamma, NA)\end{array}}{\begin{array}{l}\Gamma' = \Gamma : \alpha\,\{GL := fail : nil\} \\ \Gamma \xrightarrow{\langle Check_2, \alpha\rangle} \Gamma'\end{array}}$$

$Neut_0$ — завершение нейтрализации акторов.

$$\frac{\Gamma.\alpha\,\{GL = G\} \xrightarrow{l} \Gamma'}{\Gamma.\alpha\,\{GL = neutralize(\emptyset) : G\} \xrightarrow{l} \Gamma'}$$

$Neut_1$ — нейтрализация актора.

$$\frac{\begin{array}{l}\Gamma.\alpha\,\{Cont = \langle neutralize(\{\gamma\} \cup NA') : G, \sigma, N, B\rangle\}\,,\ \gamma \notin NA' \\ \Gamma.\gamma\,\{RL = R\}\end{array}}{\begin{array}{l}\Gamma' = \Gamma : \alpha\,\{Cont := \langle neutralize(NA') : G, \sigma, [\gamma|N], B\rangle\}\,, \\ \qquad\qquad \gamma\,\{RL := neutral : R\} \\ \Gamma \xrightarrow{\langle Neut_1, \alpha, \gamma\rangle} \Gamma'\end{array}}$$

$Succ$ — успешное завершение доказательства актора (не обязательно повторного).

$$\frac{\Gamma.\alpha\,\{GL = G\} \xrightarrow{l} \Gamma'}{\Gamma.\alpha\,\{GL = restart(\emptyset) : G\} \xrightarrow{l} \Gamma'}$$

Call — вызов повторного доказательства актора.

$$\frac{\begin{array}{l}\Gamma.\alpha\,\{FL = \langle restart(\{\gamma\} \cup RA') : G, \sigma, N, B\rangle : F\}\,,\ \gamma \notin RA' \\ \Gamma.\gamma = \langle \gamma, m(v_1, \ldots, v_k), R\rangle\end{array}}{\begin{array}{l}\Gamma' = \Gamma : \alpha \left\{ \begin{array}{l} FL := \langle wait(\gamma) : (restart(RA') : G)\,, \sigma, [\gamma|N], B\rangle \\ \quad : (\langle redo(\gamma) : (restart(RA') : G)\,, \sigma, [\gamma|N], B\rangle : F) \end{array} \right\}, \\ \qquad\qquad \gamma\,\{RL := \langle \alpha, \langle m(v_1, \ldots, v_k) : nil, \underline{\varepsilon}, [\,], [\,]\rangle : nil\rangle : R\} \\ \Gamma \xrightarrow{\langle Call, \alpha, \gamma\rangle} \Gamma'\end{array}}$$

Con — возобновление доказательства вызывающего актора (после завершения доказательства вызванного им актора).

$$\frac{\begin{array}{l}\Gamma.\alpha\,\{GL = success\} \\ \Gamma.\beta\,\{GL = wait(\alpha) : G\}\end{array}}{\begin{array}{l}\Gamma' = \Gamma : \beta\,\{GL := G\} \\ \Gamma \xrightarrow{\langle Con, \beta, \alpha\rangle} \Gamma'\end{array}}$$

Определённые в этом разделе правила взаимодействия акторов позволяют корректным образом интерпретировать в логическом языке операцию разрушающего присваивания, хотя, конечно, ради строгости изложения следует заметить, что речь идёт не о разрушении физического представления данных, а о разрушении некоторых «производных» значений общих переменных, которые можно получить, унифицировав все локальные значения активных и доказанных акторов программы (если такая унификация возможна).

Такому пониманию разрушающего присваивания в Акторном Прологе, в частности, соответствует и семантика *встроенного предиката разрушающего присваивания*. В процессе доказательства подцели вида

$$X := Y$$

осуществляется попытка унифицировать локальные значения X, Y и, затем, в случае успешной унификации — сопоставление локальных значений общих переменных программы и (если это окажется необходимым) повторное доказательство акторов, как это описано выше. Если все повторные доказательства завершаются успехом, предикат разрушающего присваивания считается истинным, и его доказательство также завершается успехом, в противном случае в логической программе происходит откат.

Заметим, что разрушающее присваивание обозначено здесь бинарным оператором *':='* лишь из соображений удобства; на основе описанного принципа можно определить операции «присваивания» любой произвольной арности.

3.4 Операционная семантика акторной машины

Операционной семантикой акторной логической машины называется отображение, ставящее в соответствие определению процедур D и начальному состоянию программы Γ^0, $\Gamma^0.\tau = \langle \tau, m(t_1, \ldots, t_k), R_\tau \rangle$, множество конечных и бесконечных цепочек меток, получаемых в результате применения определённых выше схем переходов между состояниями машины.

Определение 3.4.1. Операционная семантика $\mathcal{O}$:

$$\mathcal{O}[D, \Gamma^0] \stackrel{def}{=} \begin{array}{l} \left\{ l_1, l_2, \ldots, l_n | \Gamma^0 \xrightarrow{l_1} \Gamma_1 \xrightarrow{l_2} \ldots \xrightarrow{l_n} \Gamma_n^{SUCCESS} \right\} \cup \\ \left\{ l_1, l_2, \ldots, l_n | \Gamma^0 \xrightarrow{l_1} \Gamma_1 \xrightarrow{l_2} \ldots \xrightarrow{l_n} \Gamma_n^{FAILURE} \right\} \cup \\ \left\{ l_1, l_2, \ldots | \Gamma^0 \xrightarrow{l_1} \Gamma_1 \xrightarrow{l_2} \ldots \right\}. \end{array}$$

При этом декларативная семантика определённого @-языка соответствует декларативной семантике чистого Пролога.

Определение 3.4.2. *Исходным множеством акторных ограничений* называется множество утверждений, соответствующих доказанным акторам состояния Γ^0:

$$Init \stackrel{def}{=} \bigwedge_i M_i \text{ для всех } \langle \alpha_i, M_i, R_i \rangle \in \Gamma^0, \text{ таких что } is\_proven(\Gamma^0, \alpha_i).$$

Утверждение 3.4.3. Операционная семантика $\mathcal{O}$ корректна, то есть успешное конечное состояние программы может быть достигнуто только в том случае, если конъюнкция определения процедур D и отрицания исходного множества $Init$ и доказываемого утверждения $m(t_1, \ldots, t_k)$ невыполнима:

$$\left(\Gamma^0 \xrightarrow{\star} \Gamma^{SUCCESS}\right) \Rightarrow \left(D \cup \{\neg\,(Init\ \wedge\ m(t_1, \ldots, t_k))\} \models \bot\right).$$

Утверждение 3.4.4 (О полноте операционной семантики относительно декларативной). Если существует подстановка θ, такая что

$$D \models (Init\ \wedge\ m(t_1, \ldots, t_k))\,\theta,$$

и при исполнении программы не возникнут бесконечные вычисления, то будет достигнуто успешное конечное состояние: $\Gamma^0 \xrightarrow{\star} \Gamma^{SUCCESS}$.

Вычислительной полнотой операционная семантика $\mathcal{O}$ не обладает, поскольку реализует поиск в глубину, как в «обычном» Прологе. Программы на Акторном Прологе могут зацикливаться, однако нейтрализация акторов, сама по себе, *не* приводит к зацикливаниям — именно для того чтобы предотвратить такую опасность, был введён запрет на нейтрализацию активных акторов в схеме $Check_1$.

В формулировке утверждений 3.4.3 и 3.4.4 умышленно не упоминаются подстановки, вычисляемые при построении состояния $\Gamma^{SUCCESS}$, так как, согласно определению, акторная логическая машина завершает свою работу сразу при достижении первого же успешного состояния, и было бы некорректно говорить о «множестве вычисляемых подстановок» в целом. Впрочем, это обстоятельство нисколько не снижает важность приведённых утверждений, так как они гарантируют возможность создавать логические программы, реализующие полный перебор вариантов.

Неблагоприятным свойством акторной стратегии управления является возможность повторения большого количества дублирующих друг друга вычислений в случае, если она применяется без особой на то необходимости.

3.5 Выводы

Доказанные теоремы гарантируют *корректность логической интерпретации объектно-ориентированного программирования* в Акторном

Прологе. Однако основным математическим результатом диссертационной работы можно назвать *доказательство возможности построения корректной стратегии управления исполнением логической программы, поддерживающей повторные доказательства, параллельное исполнение, а также задержку восстановления состояний подцелей при откате*, поскольку этими свойствами обладает *механизм повторного доказательства*, реализованный в Акторном Прологе.

Формальное определение операционной семантики Акторного Пролога, построенное в этой главе на основе *последовательной абстрактной машины логического вывода*, является промежуточным звеном, помогающим корректно осуществить неформальный переход от определения Акторного Пролога к его программной реализации, однако его, конечно, не следует считать описанием реализации языка. Более того, возможности, связанные с задержкой восстановления состояний акторов при откате, можно вообще исключить из реализации — они не поддерживаются никакими синтаксическими средствами Акторного Пролога и необходимы, скорее, для того, чтобы установить, какими свойствами должна обладать стратегия управления логического языка, чтобы обеспечить корректное взаимодействие программы с «внешним миром», в котором протекают *необратимые процессы* — работа внешних устройств, заполнение полей диалоговых окон на экране и т. п.

Стратегия управления Акторного Пролога — механизм повторного доказательства — позволяет интерпретировать в логическом языке такие понятия как разрушающее присваивание и необратимый процесс и, таким образом, является решением *проблемы фрейма* [17, 27], свободным от недостатков немонотонных логических систем [33].

Необходимо также отметить, что идея локализации значений общих переменных в акторах позволяет организовать логический вывод в *распределённых системах*, в условиях противоречивости и несвоевременного обновления информации. Таким образом, разработанный логический язык является мощным средством *поддержки истинности* (см., например, [80]), обеспечивающим декларативную семантику системы акторов, инвариантную по отношению к возможному недетерминизму её поведения [21] — он является инструментом для логического программирования *открытых систем искусственного интеллекта* [37, 65].

Глава 4

Использование Акторного Пролога для анализа функциональных диаграмм информационных систем

В этой главе на примере построения и использования некоторой простой *интерактивной семантической модели* мы рассмотрим один из возможных подходов к использованию Акторного Пролога для анализа функциональных диаграмм информационных систем.

В основном, мы будем придерживаться этапов *интерактивного функционального моделирования*, рассмотренных в главе 2. Для осуществления необходимых преобразований информации мы воспользуемся интерпретатором Акторного Пролога и некоторыми другими программами, разработанными в рамках настоящей диссертационной работы:

1. Транслятор A2BIN, преобразующий исходный текст на Акторном Прологе во внутренний код интерпретатора.

2. Интерпретатор Акторного Пролога (программа PROVER).

3. Браузер Акторного Пролога (оболочка BROWSER для визуального программирования на Акторном Прологе).

4. Транслятор функциональных диаграмм IDL2BIN, преобразующий текстовое описание SADT-модели в формате *IDL* (этот формат поддерживают, например, системы Design/IDEF фирмы Meta Software

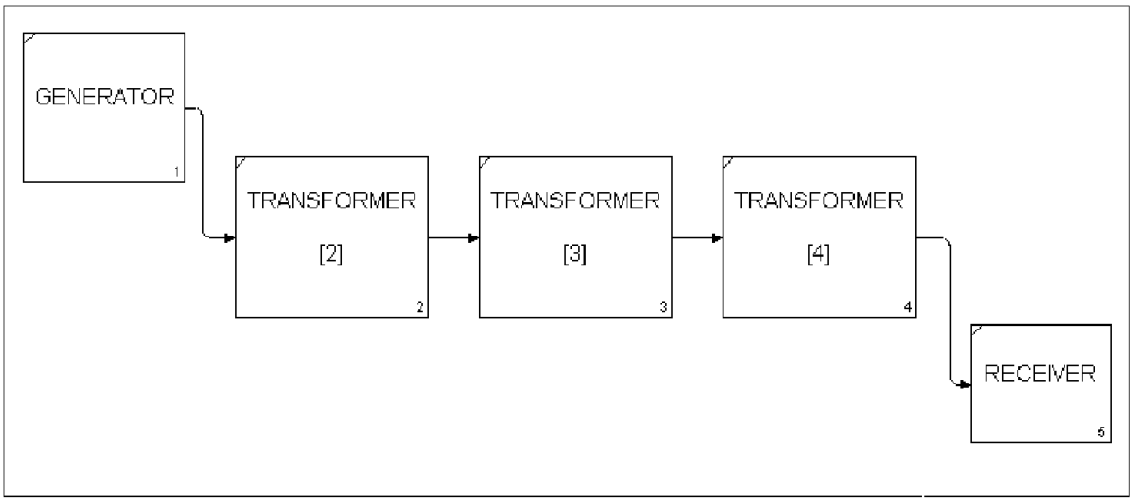


Рис. 4.1: Исходная SA-диаграмма.

и BPwin фирмы Logic Works) в программу на Акторном Прологе (в формате внутреннего кода интерпретатора).

По ходу изложения мы будем останавливаться на всех принципиальных вопросах, связанных с задачей описания компонентов функциональных диаграмм средствами логического языка. В качестве одного из возможных вариантов решения этой задачи будут рассмотрены некоторые общие принципы, а также конкретные инженерные решения, основанные на использовании различных средств Акторного Пролога.

4.1 Цель и этапы моделирования

Рассмотрим пример функциональной диаграммы (SA-диаграммы) некоторой «игрушечной» информационной системы, состоящей из пяти аппаратно-программных модулей, обрабатывающих сигналы (рис. 4.1).

Первый модуль является генератором сигналов, три других модуля — преобразователи сигналов, и последний — приёмник сигналов. Будем считать, что последовательная схема соединения блоков разработана на первом этапе моделирования и в дальнейшем изменяться не будет, однако при этом генератор и преобразователи сигналов могут использоваться в различных режимах, которым соответствуют разные пределы изменения сигналов.

Целью интерактивного функционального моделирования в рассматриваемом примере мы будем считать выбор подходящих режимов работы модулей, обеспечивающих правильность соединения отдельных модулей и правильность построения всей системы в целом.

Для достижения этой цели будут осуществлены следующие этапы интерактивного функционального моделирования:

1. Разработка классов логической программы (интерактивной семантической модели), описывающих блоки рассматриваемой функциональной диаграммы.

2. Трансляция исходной функциональной диаграммы в программу на Акторном Прологе, описывающую связи между блоками.

3. Исполнение полученной интерактивной семантической модели — выбор режимов использования модулей информационной системы в диалоге с пользователем.

В качестве результата интерактивного функционального моделирования рассматриваемой информационной системы будут получены:

1. Некоторый вариант семантически правильного соединения модулей информационной системы.

2. Логическая программа (интерактивная семантическая модель), которая может быть повторно использована (или передана другому проектировщику) для получения каких-то новых вариантов использования модулей информационной системы.

Рассмотрим названные этапы интерактивного функционального моделирования более подробно.

4.2 Описание модулей системы

В соответствии с основными принципами, изложенными в главе 2, для описания блоков функциональных диаграмм в Акторном Прологе мы воспользуемся механизмом классов.

Мы не будем поддерживать однозначное соответствие «один модуль — один класс». В рассматриваемом примере каждому модулю будет соответствовать, по крайней мере, два класса логического языка, и для этого есть несколько причин:

1. При разработке описания блоков функциональных диаграмм мы будем, по возможности, использовать принципы абстракции и наследования, выделяя общие свойства различных блоков в отдельные классы.

2. Как уже отмечалось в главе 2, в составе семантической модели целесообразно выделять части, получаемые автоматически, с помощью трансляции функциональной диаграммы, и части, разрабатываемые «вручную» — в соответствие таким частям семантической модели также удобно поставить разные классы.

В иерархии классов логической программы (рис. 4.2) будут использоваться классы, содержимое которых строится автоматически — *'DIAGRAM−A1'*, *'DIAGRAM−A2'*, *'DIAGRAM−A3'*, *'DIAGRAM−A4'*, *'DIAGRAM−A5'* (а также некоторые вспомогательные классы), и классы, разрабатываемые «вручную» (все остальные) — *'GENERATOR'*, *'TRANSFORMER'*, *'RECEIVER'* и др.

Заметим, что каждому блоку функциональной диаграммы будет соответствовать ровно один класс, создаваемый автоматически. Содержимым таких классов (в нашем примере) будет описание связей между блоками анализируемой диаграммы, хотя, на практике, конечно, автоматически могут быть построены и другие, более сложные объекты[1].

Классы, разрабатываемые «вручную», в иерархии наследования являются *предками* классов, создаваемых автоматически. Каждому блоку диаграммы, в общем случае, может соответствовать несколько таких классов, так как при их разработке проектировщик может использовать принцип абстракции, о чём говорилось выше. Например, блоку $GENERATOR$ в иерархии наследования классов (рис. 4.2) соответствует один автоматически создаваемый класс *'DIAGRAM−A1'* и два класса, разрабатываемых «вручную» — *'ALPHA'* и *'GENERATOR'* (*'ALPHA'* является предопределённым классом Акторного Пролога).

Одним из наиболее важных вопросов при построении логического описания блоков функциональной диаграммы является представление входов и выходов блоков. В нашем примере для решения этой задачи мы примем следующие соглашения:

Предположим, что некоторый класс C является составной частью описания функционального блока B.

[1]Например, программа IDL2BIN автоматически строит некоторые простейшие синтаксические тесты для связей между блоками, однако в рассматриваемом примере эти возможности нам не понадобятся.

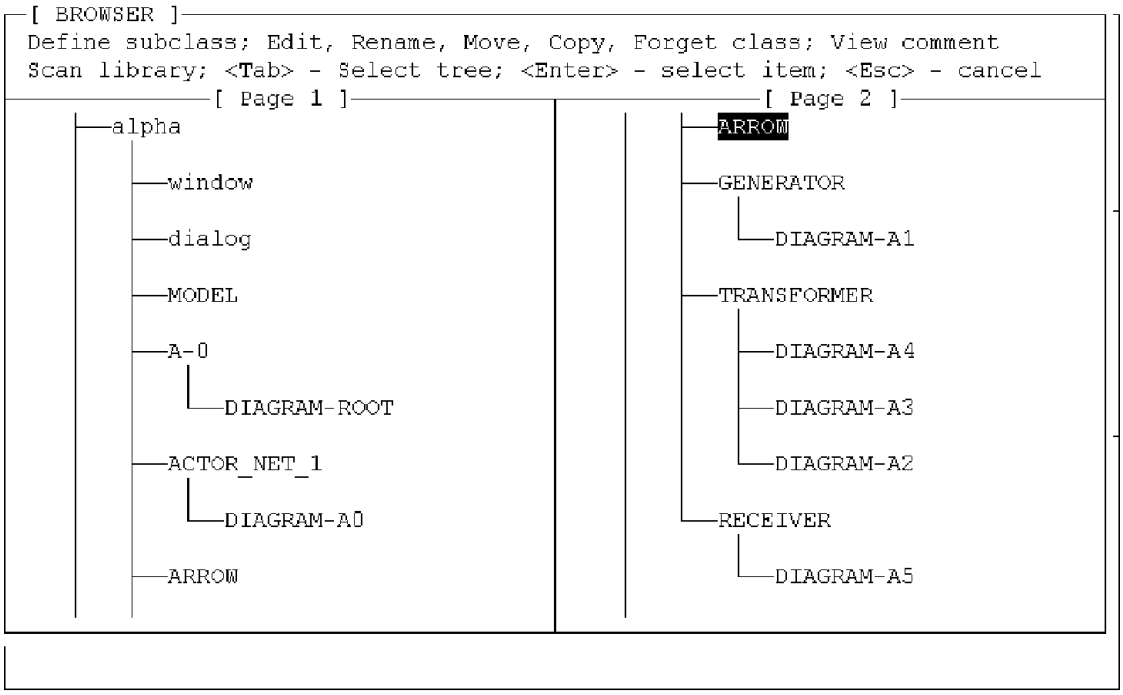


Рис. 4.2: Иерархия классов семантической модели.

1. Поставим в соответствие всем входам $I1 \ldots IK$, выходам $O1 \ldots ON$, управлению $C1 \ldots CR$ и механизмам $M1 \ldots MS$ блока B некоторые атрибуты класса C — $value\_i1 \ldots value\_iK$, $value\_o1 \ldots value\_oN$, $value\_c1 \ldots value\_cR$, $value\_m1 \ldots value\_mS$, обозначающие данные. Пользуясь этим соглашением, мы будем интерпретировать передачу информации через какую-либо из перечисленных интерфейсных точек блока B как изменение значения слота, обозначенного соответствующим атрибутом. Например, передача сообщения $Text$ через выход $O7$ блока B может быть представлена в синтаксисе Акторного Пролога как

$$value\_o7 := Text.$$

2. Кроме того, поставим в соответствие всем указанным выше входам, выходам, управлению и механизмам блока B атрибуты вида $entry\_i1 \ldots entry\_iK$, $entry\_o1 \ldots entry\_oN$, $entry\_c1 \ldots entry\_cR$, $entry\_m1 \ldots entry\_mS$, обозначающие некоторые экземпляры классов. Пользуясь таким соглашением, можно интерпретировать передачу информации через какую-либо из интерфейсных точек блока B как дальний вызов предиката в мире, обозначенном соответствующим атрибутом. Например, передача сообщения $message(Text)$ через выход $O7$ блока B может быть представлена как

$$entry\_o7 \ ?\ message(Text).$$

3. Получение информации блоком B извне мы будем интерпретировать, соответственно, как повторное доказательство акторов, зависящих от значений слотов $value\_X$ (если для передачи сообщения используется соглашение 1) или как вызов предиката в мире, соответствующем моделируемому блоку (если для передачи сообщения используется соглашение 2).

   Во втором случае, для того чтобы иметь возможность различать, через какую именно интерфейсную точку пришло сообщение, мы будем использовать строго определённый формат предикатов для описания заголовков предложений, обрабатывающих принимаемую информацию. Конкретно, мы будем использовать в заголовках предложений недоопределённые множества следующего вида:

   - $"iN"\{|Rest\}$ — для информации, принимаемой через входы (N — это целое число, номер интерфейсной точки, а $Rest$ — «информационная часть» принятого сообщения);

- "oN"$\{|Rest\}$ — для информации, принимаемой через выходы[2];
- "cN"$\{|Rest\}$ — для управляющей информации;
- "mN"$\{|Rest\}$ — для исполнителей.

4. В соответствии с соглашением (3), соглашение (2) может быть несколько уточнено, то есть для построения дальних вызовов предикатов мы также будем использовать недоопределённые множества. Например, с учётом этого требования, передача сообщения через выход $O7$ блока B будет выглядеть следующим образом:

$$entry\_o7\ ?\ message\{symbol : Text\}.$$

Ниже приведено содержимое классов, разработанных проектировщиком «вручную». Классы, созданные автоматически, будут рассмотрены в следующем разделе.

```
------------------------------------------------------------
-- Основной класс                                         --
------------------------------------------------------------
class 'MODEL' using 'DIALOG' is
--
comment= "Пример интерактивной семантической модели"
--
box= ('DIAGRAM-ROOT',   -- Слот box содержит экземпляр
       host= self)      -- класса 'DIAGRAM-ROOT'.
name= "MAIN-BOX"        -- Имя диалога.
[
goal:-!,
       show.            -- Разворачивание окна диалога.
--
event(5001):-!,         -- Функции экранных кнопок:
       break.           -- "EXIT" - выход из программы;
event(5002):-!,         -- "HELP" - подсказка.
       ('DIALOG',name= "HELP") ? show.
]
```

[2]Разумеется, в процессе логического анализа может понадобиться передавать информацию через любые интерфейсные точки, в любых направлениях — даже таких, которые никак не соответствуют реальной передаче данных в моделируемой системе.

```
------------------------------------------------------------
-- Описание блока "Генератор"                              --
------------------------------------------------------------
class 'GENERATOR' using 'DIALOG' is
--
entry_o1                -- Блок GENERATOR имеет один
value_o1                -- выход - O1.
--
number  -- Значения слотов number (номер блока), x и y
x       -- (координаты блока на экране) будут заданы
y       -- в автоматически создаваемом классе 'DIAGRAM-A1'.
--
name= "GENERATOR-BOX"   -- Имя диалога.
--
power   -- Слот power обозначает состояние генератора:
        -- 1 - включён, 0 - выключен.
[
goal:-!,
        move(x,y),      -- Размещение окна диалога
                        -- в нужной точке экрана.
        @ accept_generator_state,
        show.           -- Разворачивание окна диалога.
--
event(1):-              -- Функция экранной кнопки "SET".
        & accept_generator_state.
event(_):-              -- В случае неудачи -
        beep.           -- звуковой сигнал.
--
accept_generator_state:-
        -- Ввод значения экранного переключателя.
        get(7050,power),
        power == 0,!,   -- Проверка значения.
        color(95,95).   -- Перекрашивание окна диалога в
                        -- фиолетовый цвет.
accept_generator_state:-
        get(7050,power),
        -- Ввод значения поля редактирования "Напряжение".
        get(7030,value_o1),
        color(47,47).   -- Перекрашивание в зелёный цвет.
```

```
--
"o1"{check_power:ok}:-  -- Обработка специального
        copy(power),    -- сообщения для проверки
        power == 1,!.   -- состояния генератора.
"o1"{check_power:failure}:-
        copy(power).
]
-----------------------------------------------------------
-- Описание блока "Преобразователь"                       --
-----------------------------------------------------------
class 'TRANSFORMER' using 'DIALOG' is
--
entry_i1                -- Блок TRANSFORMER имеет две
value_i1                -- интерфейсные точки -
entry_o1                -- вход I1 и выход O1.
value_o1
--
number  -- Значения слотов number, x и y будут заданы
x       -- в автоматически создаваемых классах
y       -- 'DIAGRAM-A2', DIAGRAM-A3', DIAGRAM-A4'.
--
name= "TRANSFORMER-BOX" -- Имя диалога.
--
mode    -- Слот mode обозначает номер режима, в котором
        -- используется преобразователь.
--
dialog= ('TRANSFORMER_DIALOG',
        sw= mode,
        label= number)
--
flag    -- Слот flag обозначает, согласован ли
        -- преобразователь со своими соседями:
        -- значение ok - да, failure - нет.
[
goal:-!,
        move(x,y),      -- Размещение окна диалога.
        @ check_state,  -- Построение актора.
        show.           -- Разворачивание окна диалога.
--
```

```
check_state:-           -- Проверка режима.
        in(value_i1),   -- Получение информации о
        in(value_o1),   -- режиме устройства (mode) и
        in(mode),       -- режимах его соседей.
        select_mode(mode,value_i1,value_o1),
        put(7001,value_i1),
        put(7002,value_o1),
        flag := ok,!,   -- Изменение значения слота flag.
        color(47,47).   -- Перекрашивание в зелёный цвет.
check_state:-           -- В случае неудачи:
        copy(value_i1), -- Установление логической
        copy(value_o1), -- связи с общими переменными.
        copy(mode),
        display_input_value,
        display_output_value,
        -- Изменение значений слотов mode и flag.
        mode := 0,
        flag := failure,
        -- Перекрашивание окна диалога в фиолетовый цвет.
        color(95,95).
--
-- Таблица режимов преобразователей. Первый параметр - номер
-- режима, второй - входное напряжение, третий - выходное.
--
select_mode(1,220,110).
select_mode(2,220,36).
select_mode(3,110,12).
select_mode(4,36,12).
select_mode(5,36,110).
select_mode(6,12,220).
--
display_input_value:-
        bound(value_i1),
        put(7001,value_i1).
display_input_value:-
        value_i1 == 0,
        put(7001,"---").
--
```

```
display_output_value:-
        bound(value_o1),!,
        put(7002,value_o1).
display_output_value:-
        value_o1 == 0,
        put(7002,"---").
--
event(1):-!,            -- Обработка нажатия кнопки "TUNE".
        dialog ? show.  -- Разворачивание окна диалога.
--
-- Обработка специального сообщения для проверки состояния
-- преобразователя.
--
"o1"{check_power:Answer2}:-
        entry_i1 ? message{check_power:Answer1},
        copy(flag,Answer1),
        answer(flag,Answer1,Answer2).
--
answer(ok,ok,ok):-!.    -- В случае если состояние устройства,
answer(_,_,failure).    -- а также его соседа слева - "ok",
                        -- предикат возвращает значение "ok".
]
-------------------------------------------------------------
-- Описание блока "Приёмник"                                --
-------------------------------------------------------------
class 'RECEIVER' using 'DIALOG' is
--
entry_i1                -- Блок RECEIVER имеет один
value_i1                -- вход - I1.
--
number  -- Значения слотов number, x и y будут заданы
x       -- в автоматически создаваемом классе
y       -- 'DIAGRAM-A5'.
--
name= "RECEIVER-BOX"    -- Имя диалога.
--
error_message= ('DIALOG',
        name= "RECEIVER-ERROR")
```

```
[
goal:-!,
        -- Размещение окон диалогов на экране.
        move(x,y),
        error_message ? move(x,y),
        @ check_state,  -- Построение акторов.
        @ check_whole_system.
--
check_state:-           -- Проверка "напряжения на входе".
        copy(value_i1), -- Приём значения value_i1.
        check_integer(value_i1),
        put(8002,value_i1).
--
check_integer(0):-!.    -- Означивание несвязанных
check_integer(_).       -- переменных нулём.
--
-- Проверка режимов всех устройств в цепочке.
--
check_whole_system:-
        copy(value_i1), -- Приём значения value_i1.
        entry_i1 ? message{check_power:Answer},
        copy(Answer),!, -- Приём значения Answer.
        display_message(Answer).
--
-- Отображение результатов проверки на экране.
-- Если в цепочке устройств "всё в порядке", разворачивается
-- окно диалога 'RECEIVER-BOX' (зелёного цвета). Иначе -
-- окно диалога 'RECEIVER-ERROR' (красного цвета).
--
display_message(failure):-
        hide,
        error_message ? show.
display_message(ok):-
        show,
        error_message ? hide.
]
```

```
------------------------------------------------------------
-- Вспомогательный класс для выбора режима преобразователя  --
------------------------------------------------------------
class 'TRANSFORMER_DIALOG' using 'DIALOG' is
name= "TRANSFORMER-PARAMETERS"
sw      -- Слот sw обозначает номер выбранного режима.
label   -- Слот label - вспомогательный - хранит номер блока.
[
goal:-!,
        -- Ввод значения экранного переключателя.
        put(7200,label).
--
event(1):-!,            -- Кнопка "OK".
        & accept_transformer_state,
        hide.           -- Сворачивание окна диалога.
event(2):-              -- Кнопка "CANCEL".
        hide.
--
-- Временный актор устанавливает логическую связь между
-- значением экранного переключателя и слотом sw.
--
accept_transformer_state:-
        get(7100,sw).
accept_transformer_state:-
        beep,
        fail.
]
------------------------------------------------------------
```

Описание экранных форм, используемых предопределённым классом *'DIALOG'* для организации интерфейса с пользователем, подготавливается в специальном формате в отдельном файле и загружается в процессе работы программы PROVER.

```
------------------------------------------------------------
--          Пример файла описания экранных форм             --
------------------------------------------------------------
dialog(modeless,"MAIN-BOX",0,0,80,1,127,0,"")
        buttons
                cancel(5001,1,0," EXIT ",2,0)
```

```
                f(5002,11,0," HELP ",2,1)
        end_of_buttons
end_of_dialog
dialog(modeless,"HELP",3,3,43,11,31,31,"")
        buttons
                cancel(0,1,8,"  OK  ",3,0)
        end_of_buttons
        string(0,0," THIS ACTOR SYSTEM . . . ")
        . . .
        string(0,7,"----------------------------------------")
end_of_dialog
dialog(modeless,"GENERATOR-BOX",0,0,22,7,111,111,"")
        string(1,0,"STATE OF GENERATOR")
        checkbox(7050,16,1,string(1,1,"POWER (OFF/ON)"),0)
        string(1,2,"VOLTAGE")
        editint(7030,14,2,3,0)
        string(0,3,"------------------")
        buttons
                ok(0001,1,4," SET ",2,0)
        end_of_buttons
end_of_dialog
dialog(modeless,"TRANSFORMER-BOX",0,0,21,7,111,111,"")
        string(1,0,"TRANSFORMER")
        string(8,2,"x")
        text(7001,1,2,3,"0")
        text(7002,11,2,3,"0")
        string(0,3,"------------------")
        buttons
                ok(0001,1,4," TUNE ",2,0)
        end_of_buttons
end_of_dialog
dialog(modeless,"TRANSFORMER-PARAMETERS",25,3,35,9,63,63,"")
        buttons
                ok(0001,1,6,"  OK  ",3,0)
                cancel(0002,11,6,"CANCEL",1,0)
        end_of_buttons
        string(0,5,"--------------------------------")
        rbuttons(7100,00001)
                rb(1,2,string(4,2," 220 / 110"))
```

```
                rb(1,3,string(4,3," 220 / 36"))
                rb(1,4,string(4,4," 110 / 12"))
                rb(18,2,string(21,2," 36 / 12"))
                rb(18,3,string(21,3," 36 / 110"))
                rb(18,4,string(21,4," 12 / 220"))
        end_of_rbuttons
        string(0,1,"--------------------------------")
        string(1,0,"DEVICE")
        text(7200,8,0,3,"0")
end_of_dialog
dialog(modeless,"RECEIVER-BOX",0,0,18,7,111,111,"")
        buttons
                cancel(0,1,4,"  OK  ",3,0)
        end_of_buttons
        string(1,0,"RECEIVER")
        text(8002,1,2,3,"0")
end_of_dialog
dialog(modeless,"RECEIVER-ERROR",55,3,19,7,79,79,"")
        buttons
                cancel(0,1,4,"  OK  ",3,0)
        end_of_buttons
        string(0,1,"    POWER OFF    ")
        string(0,3,"-----------------")
end_of_dialog
```

4.3 Трансляция SADT-модели

Ещё одним принципиальным вопросом, связанным с логической интерпретацией функциональных диаграмм, является представление связей между блоками диаграммы. В нашем примере при условии соблюдения соглашений о представлении интерфейсных точек блоков, принятых в разделе 4.2, для решения этой задачи понадобится каким-то образом преодолеть следующее противоречие.

Предположим, что некоторые функциональные блоки B_1 и B_2 связаны между собой, причём связь проведена между выходом $O1$ блока B_1 и входом $I1$ блока B_2 (рис. 4.3). В соответствии с соглашением (3) раздела 4.2, сообщения из блока B_1 должны приходить в блок B_2 в виде дальних вызовов предикатов $"i1"\{|Rest\}$, однако проектировщик блока B_1, в общем

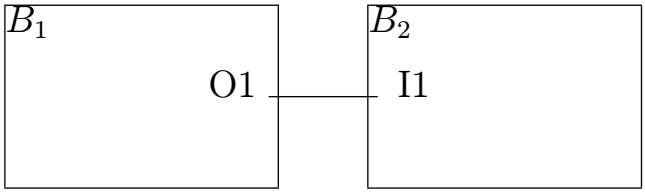


Рис. 4.3: Связь между блоками одного уровня вложенности.

случае, не может знать заранее, в каких именно связях будет участвовать этот блок в функциональной диаграмме, и следовательно, от него нельзя требовать, чтобы он все обращения к выходу $O1$ оформлял в виде вызовов предиката

$$entry\_o1\ ?\ "i1"\{|Rest\}.$$

Для решения этой проблемы в рассматриваемом примере мы воспользуемся следующим подходом. Передачу информации через выход $O1$ блока B_1 (или, по аналогии, через какую-то другую интерфейсную точку) мы будем осуществлять с помощью дальнего вызова предиката

$$entry\_o1\ ?\ message\{|Rest\},$$

а добавление компонента *"i1"* в передаваемое множество $\{|Rest\}$ будет происходить автоматически, по правилам, генерируемым в процессе трансляции функциональной диаграммы.

Конкретно, для реализации такой «переадресации» сообщений, все слоты *entry_X* соответствующих экземпляров классов семантической модели будут заполняться экземплярами вспомогательного класса *'ARROW'*, текст которого будет автоматически добавлен в состав программы в процессе трансляции функциональной диаграммы:

```
class 'ARROW' is
host
entry
[
goal.
--
message{|REST}:-
        host ? {'':entry|REST}.
]
```

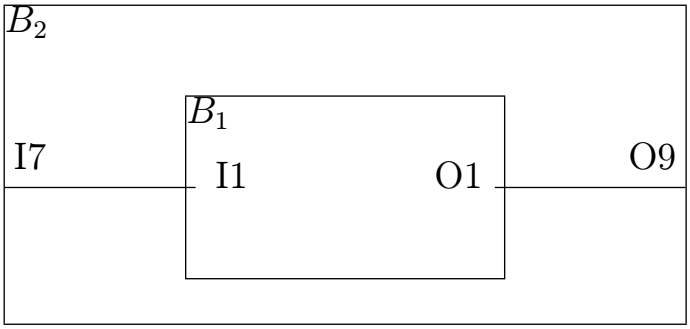


Рис. 4.4: Связь между блоками разных уровней вложенности.

Например, в процессе трансляции функциональной диаграммы на рис. 4.3 в состав конструктора экземпляра класса $C1$, соответствующего блоку B_1, будет автоматически добавлено определение выхода $O1$:

```
box_1= ('C1',
        entry_o1= ('ARROW',
                host= self,
                entry= "1o1"),
        value_o1= CommonVariable)
```

При этом в состав предложений класса, описывающего соединение между блоками B_1 и B_2, будет включено «правило переадресации» сообщений, имитирующее логику второго порядка:

```
"1o1"{|REST}:-
        box_2 ? "i1"{|REST}.
```

(единица перед символом o в строке "1o1" — это номер блока B_1 на диаграмме). В результате, все сообщения, передаваемые через выход $O1$ блока B_1, будут отправляться в мир box_2 (соответствующий блоку B_2 в нашем примере), снабженные «меткой» "i1".

Аналогичная проблема возникает и при передаче сообщений между блоками разных уровней вложенности SA-диаграмм. Для реализации передачи сообщений между интерфейсными точками SA-диаграммы и блоками, заданными в её составе (см., например, случай на рис. 4.4), также приходится использовать специальные предложения, (автоматически) генерируемые в процессе трансляции диаграмм. В частности, для представления связей, изображённых на рис. 4.4, можно использовать следующие «правила второго порядка»:

```
"i7"{|REST}:-
        box_1 ? "i1"{|REST}.
"1o1"{|REST}:-
        entry_o9 ? message{|REST}.
```

Ниже приведены классы семантической модели, автоматически построенные в процессе трансляции функциональной диаграммы 4.1.

```
-------------------------------------------------------------
-- Целевое утверждение программы                            --
-------------------------------------------------------------
project is
        ('MODEL',
                project_name= "Model 2",
                title= "ACTOR NET 1",
                language_version= "1.2.8")
-------------------------------------------------------------
class 'DIAGRAM-ROOT' using 'A-0' is
--
box_0= ('DIAGRAM-A0')   -- Экземпляр класса 'DIAGRAM-A0'
--                      -- используется в классе 'MODEL',
number= root            -- созданном "вручную".
[
]
-------------------------------------------------------------
class 'A-0' using 'ALPHA' is
[                       -- Вспомогательный класс.
]                       -- Может быть достроен "вручную".
-------------------------------------------------------------
--
-- Экземпляр этого класса используется в автоматически
-- созданном классе 'DIAGRAM-ROOT'.
--
class 'DIAGRAM-A0' using 'ACTOR_NET_1' is
--
link_1
link_2                  -- Вспомогательные общие переменные.
link_3
link_4
--
```

```
box_1= ('DIAGRAM-A1',
        entry_o1= ('ARROW',
                host= self,
                entry= "1o1",
                value= link_4),
        value_o1= link_4)
box_2= ('DIAGRAM-A2',
        entry_i1= ('ARROW',
                host= self,
                entry= "2i1",
                value= link_4),
        value_i1= link_4,
        entry_o1= ('ARROW',
                host= self,
                entry= "2o1",
                value= link_3),
        value_o1= link_3)
box_3= ('DIAGRAM-A3',
        entry_i1= ('ARROW',
                host= self,
                entry= "3i1",
                value= link_3),
        value_i1= link_3,
        entry_o1= ('ARROW',
                host= self,
                entry= "3o1",
                value= link_2),
        value_o1= link_2)
box_4= ('DIAGRAM-A4',
        entry_i1= ('ARROW',
                host= self,
                entry= "4i1",
                value= link_2),
        value_i1= link_2,
        entry_o1= ('ARROW',
                host= self,
                entry= "4o1",
                value= link_1),
        value_o1= link_1)
```

```
box_5= ('DIAGRAM-A5',
        entry_i1= ('ARROW',
                host= self,
                entry= "5i1",
                value= link_1),
        value_i1= link_1)
--
number= 0
x= 12                   -- Эти значения могут использоваться
y= 4                    -- в классе 'ACTOR_NET_1'.
color= 30
[
"1o1"{|REST}:-!,
        box_2 ? "i1"{|REST}.
"2i1"{|REST}:-!,
        box_1 ? "o1"{|REST}.
"3o1"{|REST}:-!,
        box_4 ? "i1"{|REST}.
"4i1"{|REST}:-!,
        box_3 ? "o1"{|REST}.
"2o1"{|REST}:-!,
        box_3 ? "i1"{|REST}.
"3i1"{|REST}:-!,
        box_2 ? "o1"{|REST}.
"4o1"{|REST}:-!,
        box_5 ? "i1"{|REST}.
"5i1"{|REST}:-!,
        box_4 ? "o1"{|REST}.
]
-------------------------------------------------------------
class 'ACTOR_NET_1' using 'ALPHA' is
[ ]                     -- Вспомогательный класс.
-------------------------------------------------------------
class 'DIAGRAM-A1' using 'GENERATOR' is
number= 1
x= 2                    -- Эти значения используются
y= 8                    -- в классе 'GENERATOR',
color= 47               -- созданном "вручную".
[ ]
```

```
------------------------------------------------------------
class 'DIAGRAM-A2' using 'TRANSFORMER' is
number= 2
x= 16                   -- Эти значения используются
y= 13                   -- в классе 'TRANSFORMER',
color= 47               -- созданном "вручную".
[ ]
class 'DIAGRAM-A3' using 'TRANSFORMER' is
number= 3
x= 33                   -- Эти значения используются
y= 13                   -- в классе 'TRANSFORMER',
color= 47               -- созданном "вручную".
[ ]
class 'DIAGRAM-A4' using 'TRANSFORMER' is
number= 4
x= 51                   -- Эти значения используются
y= 13                   -- в классе 'TRANSFORMER',
color= 47               -- созданном "вручную".
[ ]
------------------------------------------------------------
class 'DIAGRAM-A5' using 'RECEIVER' is
number= 5
x= 68                   -- Эти значения используются
y= 17                   -- в классе 'RECEIVER',
color= 47               -- созданном "вручную".
[ ]
------------------------------------------------------------
class 'ARROW' using 'ALPHA' is
host
entry
value
[
goal:-!.
message{|REST}:-!,
        host ? {'':entry|REST}.
link{|REST}:-!,
        value == REST.
]
------------------------------------------------------------
```

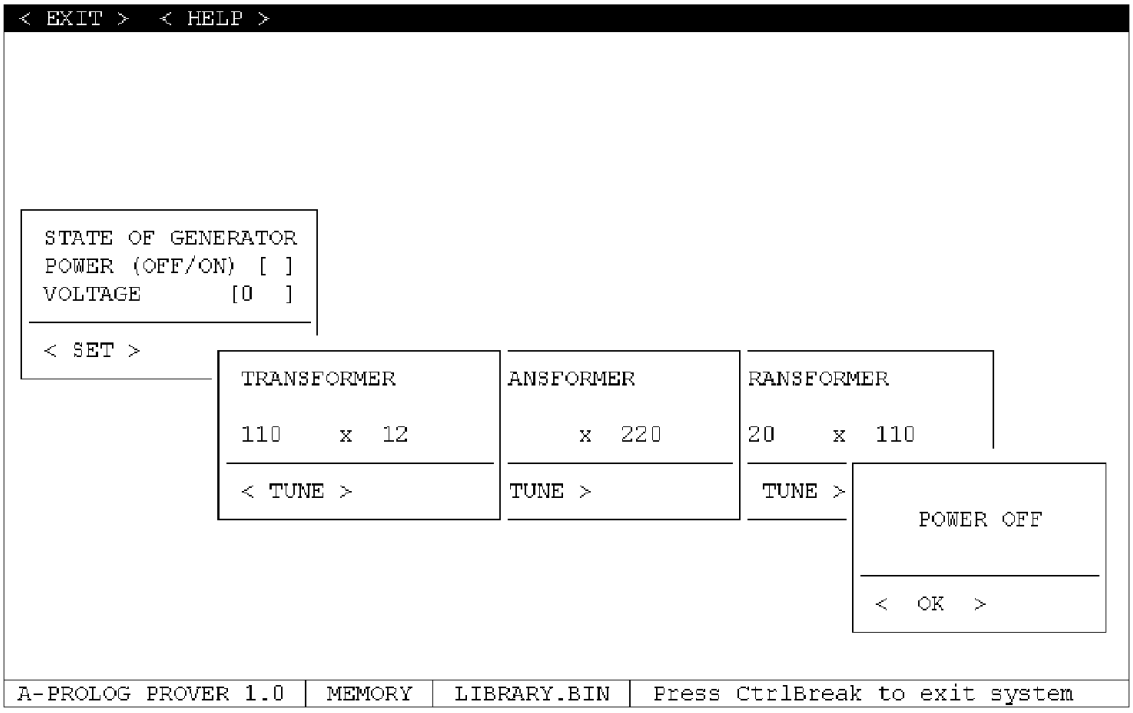


Рис. 4.5: Визуальный интерфейс интерактивной семантической модели.

4.4 Исполнение логической программы

После того как построены все необходимые классы интерактивной семантической модели, она может быть исполнена.

В нашем примере созданная логическая программа поддерживает визуальный интерфейс с пользователем (рис. 4.5), в котором каждому блоку исходной функциональной диаграммы соответствует некоторый прямоугольник с кнопками. Пользователь имеет возможность изменять параметры компонентов модели (рис. 4.6).

В программе каждому блоку диаграммы поставлен в соответствие логический актор, отслеживающий режимы работы ближайших модулей, соединённых с рассматриваемым, и меняющий цвет «своего» прямоугольника на экране в зависимости от того, согласуются ли эти режимы с режимом его собственного блока. Блок, согласованный со своими соседями, выделяется зелёным цветом.

В случае если текущему режиму некоторого блока не соответствуют режимы соседних блоков, его актор пытается автоматически подобрать

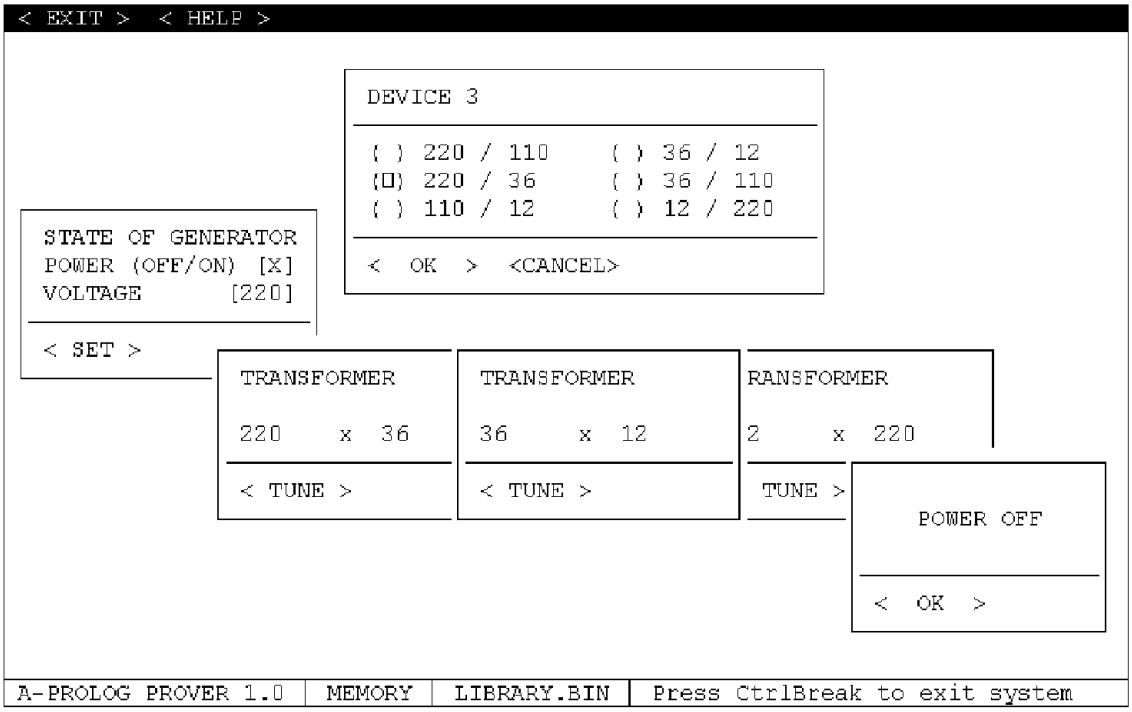


Рис. 4.6: Диалог пользователя с интерактивной семантической моделью.

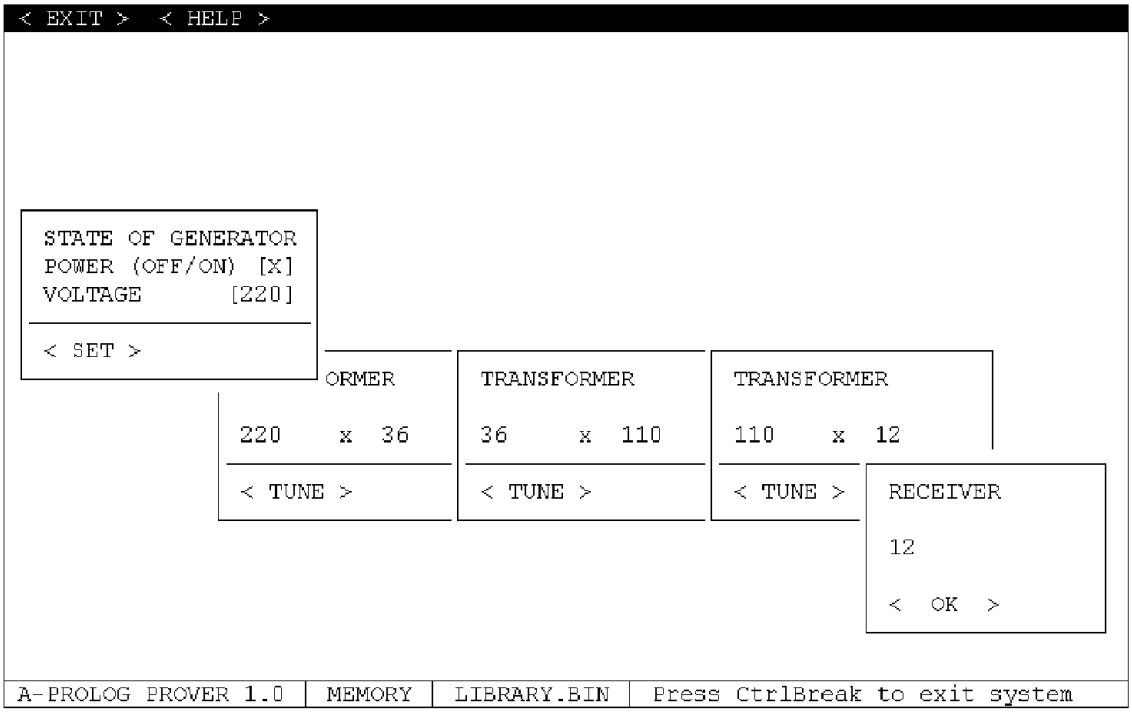


Рис. 4.7: Результат диалога — семантически правильное соединение компонентов модели.

для него другой подходящий режим или, если это ему не удаётся, перекрашивает соответствующий прямоугольник на экране.

Кроме того, актор блока $RECEIVER$ отслеживает режимы работы всех блоков в цепи передачи сигналов и меняет цвет «своего» прямоугольника в зависимости от того, все ли блоки в цепи «генератор–приёмник» согласованы между собой. Если все блоки согласованы, и генератор находится в режиме «включён», блок $RECEIVER$ перекрашивается в зелёный цвет, в противном случае — в красный.

Результатом работы пользователя с интерактивной семантической моделью должен стать некоторый вариант (или несколько вариантов) выбора режимов работы модулей информационной системы, обеспечивающий семантически правильное соединение этих модулей (рис. 4.7).

4.5 Выводы

Поддержка в Акторном Прологе средств ООП позволяет использовать преимущества ООП при проектировании и семантическом анализе функциональных диаграмм, а именно:

- Позволяет локализовать и «спрятать» составные части семантической модели в отдельные блоки, так чтобы затем эти блоки можно было использовать в процессе разработки функциональной диаграммы в качестве «чёрных ящиков», не вникая в их логическую «начинку».

- Позволяет использовать при разработке семантических моделей функциональных диаграмм принципы *абстракции* и *наследования* (а также допускает *повторное использование описаний* функциональных блоков — в других диаграммах).

- Позволяет в значительной степени автоматизировать разработку семантических моделей с помощью *автоматической трансляции графических функциональных диаграмм* в текст на объектно-ориентированном логическом языке.

- Позволяет реализовать *интерактивный режим работы* с проектировщиком или читателем функциональной диаграммы.

Разумеется, рассмотренный в этой главе пример не способен в полной мере показать все возможности использования средств и методов объектно-ориентированного логического программирования для интерактивного функционального моделирования, связанные, например, с осуществлением интерактивного редактирования структуры функциональных диаграмм в процессе исполнения семантических моделей или другими идеями из области визуализации данных и человеко-машинного интерфейса. Заметим лишь, что практическая реализация многих из них в большой степени зависит от развития и разработки новых интерфейсных средств логического языка.

Заключение

В диссертационной работе получены следующие результаты:

1. Впервые предложен и исследован *метод логического объектно-ориентированного описания и анализа функциональных диаграмм информационных систем*, обеспечивающий семантическую правильность функциональных диаграмм в процессе интерактивного проектирования.

2. Впервые предложена и разработана *логическая акторная модель функциональной диаграммы, проектируемой в интерактивном режиме*, представляющая диаграмму в виде теоремы на логическом языке, разделённой на повторно доказываемые подцели (логические акторы), взаимодействующие в объектно-ориентированном пространстве поиска.

3. Впервые предложен и разработан *метод повторного доказательства подцелей логической программы*, обеспечивающий корректность доказательства теоремы при произвольном порядке доказательства подцелей и позволяющий интерпретировать в логическом языке интерактивный режим, разрушающее присваивание и необратимые процессы.

4. Впервые предложена и разработана *логическая интерпретация объектно-ориентированного программирования* на основе метода повторного доказательства подцелей. Разработана *парадигма программирования*, соответствующая предложенной логической интерпретации объектно-ориентированного программирования.

5. Разработан *объектно-ориентированный логический язык* — Áкторный Пролог. Построено полное *определение синтаксиса и семантики* Акторного Пролога.

6. Доказаны *теоремы, гарантирующие корректность логической интерпретации объектно-ориентированного программирования* в Акторном Прологе.

7. Построено *формальное определение операционной семантики* последовательной версии Акторного Пролога без средств управления.

8. Реализованы *прототипы интерпретатора*, *браузера* Акторного Пролога, а также *транслятора* в исходный текст на языке PDC PROLOG. Показана возможность эффективной реализации Акторного Пролога на персональных ЭВМ.

Литература

[1] **Алиев Т. М., Тер-Хачатуров А. А.** Измерительная техника. — М.: Высшая школа, 1991. — 384 с.

[2] **Антимиров В. М., Воронков А. А., Дегтярёв А. И., Захарьящев М. В., Проценко В. С.** Математическая логика в программировании. Обзор // Математическая логика в программировании / Под ред. М. В. Захарьящева и Ю. И. Янова. — М.: Мир, 1991. — С. 331–407.

[3] **Бобровский С.** Комтек 97: тенденции развития CASE-систем // PC WEEK / RE. — 1997. — 3 июня. — С. 40–41.

[4] **Буч Г.** Объектно-ориентированное проектирование с примерами применения. — М.: Конкорд, 1992. — 519 с.

[5] **Галаган Н. И., Румянцев Ю. И., Адеев В. В., Бобков В. О., Мазан В. В.** Инструментальная система логического программирования на основе графических способов визуализации процессов программирования, анализа и отладки //Управляющие системы и машины. — 1992. — № 5/6. — С. 42–50.

[6] **Гуляев Ю. В., Олейников А. Я., Филинов Е. Н.** Развитие и применение открытых систем в Российской Федерации // Информационные технологии и вычислительные системы. — 1995. — Т. 1. — № 1. — С. 1–12.

[7] **Джонс Дж. К.** Методы проектирования. — М.: Мир, 1986. — 326 с.

[8] **Зиндер Е. З.** Проектирование баз данных: новые требования, новые подходы // СУБД. — 1996. — № 3. — С. 10–22.

[9] **Ивлев В., Попова Т., Огороднийчук Д.** Использование CASE-средств для совершенствования деятельности предприятий // PC WEEK / RE. — 1997. — 23 сентября. — С. 53–54.

[10] **Ин Ц., Соломон Д.** Использование Турбо-Пролога. — М.: Мир, 1993. — 608 с.

[11] **Калянов Г. Н.** CASE структурный системный анализ (автоматизация и применение). — М.: Лори, 1996. — 242 с.

[12] **Керов Л., Кучуков А.** Система логического программирования PDC PROLOG // Soft Review / Компьютерное обозрение. — 1993. — № 12. — С. 12–13; 1994. — № 1. — С. 20–24.

[13] **Кларк К., Маккейб Ф., Грегори С.** Средства языка IC-Prolog // Логическое программирование / Под ред. В. Н. Агафонова. — М.: Мир, 1988. — С. 245–260.

[14] **Клоксин У., Меллиш К.** Программирование на языке Пролог. — М.: Мир, 1987.

[15] **Колмероэ А., Кануи А., ван Канегем М.** Пролог — теоретические основы и современное развитие // Логическое программирование / Под ред. В. Н. Агафонова. — М.: Мир, 1988. — С. 27–133.

[16] Логика и компьютер. Моделирование рассуждений и проверка правильности программ / Н. А. Алешина, А. М. Анисов, П. И. Быстров и др. — М.: Наука, 1990. — 240 с.

[17] **Маккарти Д.** Общность в системах искусственного интеллекта // Лекции лауреатов премии Тьюринга / Под ред. Р. Эшенхёрста. — М.: Мир, 1993. — С. 299–312.

[18] **Малпас Дж.** Реляционный язык Пролог и его применение. — М.: Наука, 1990. — 464 с.

[19] **Марка Д., МакГоуэн К.** Методология структурного анализа и проектирования. — М.: МетаТехнология, 1993. — 240 с.

[20] **Миллер Д.** Логический анализ модулей в логическом программировании // Математическая логика в программировании / Под ред. М. В. Захарьящева и Ю. И. Янова. — М.: Мир, 1991. — С. 233–273.

[21] **Морозов А. А., Обухов Ю. В., Олейников А. Я.** Логическое программирование открытых систем // Логика, методология, философия науки: Тез. докл. XI межд. конф. — Обнинск, 1995. — Т. 2. — С. 153–156.

[22] **Морозов А. А., Обухов Ю. В.** Акторный Пролог. Определение языка программирования. — Москва, 1996. — Препринт ИРЭ РАН 2(613) от 14.06.96. — 57 с.
(http: // www.cplire.ru / Lab144 / index.html)

[23] **Морозов А. А., Обухов Ю. В.** Логическая акторная модель прикладных открытых систем // Развитие и применение открытых систем: Тез. докл. II межд. конф. — Петрозаводск, 1995. — С. 28–30.

[24] **Морозов А. А., Обухов Ю. В.** Семантический анализ функциональных диаграмм информационных систем средствами объектно-ориентированного логического программирования // Развитие и применение открытых систем: Тез. докл. IV межд. конф. — Нижний Новгород, 1997. — С. 61–64.
(http: // www.rapros97.nnov.ru / reports / 9.html)

[25] **Морозов А. А.** Акторный Пролог // Дискретные модели в теории управляющих систем: Тез. докл. II межд. конф. — М.: Диалог-МГУ, 1997. — С. 42.
(http: // www.cplire.ru / Lab144 / report1.html)

[26] **Морозов А. А.** Акторный Пролог // Программирование. — 1994. — № 5. — С. 66–78.

[27] **Морозов А. А.** Решение проблемы фрейма в Акторном Прологе // Прикладная логика — 95: Тез. докл. IV межд. конф. — Иркутск, 1995. — С. 55–56.

[28] Развивающаяся объектно-ориентированная система Smalltalk-80 // Программное обеспечение персональных ЭВМ / Под ред. А. А. Стогния. — Киев: Наук. Думка, 1989. — С. 278–354.

[29] **Робинсон Дж.** Логическое программирование — прошлое, настоящее и будущее // Логическое программирование / Под ред. В. Н. Агафонова. — М.: Мир, 1988. — С. 7–26.

[30] **Сбытов Н. Н., Трубицын А. В.** Окно в новое измерение // Мир ПК. — 1993. — № 7. — С. 71–75.

[31] **Страуструп Б.** Язык программирования C++: В 2 т. — Киев: ДиаСофт, 1993.

[32] **Сухомлин В. А.** Концепция Глобальной информационной инфраструктуры (GII) и её взаимосвязь с концепцией открытых систем // Развитие и применение открытых систем: Тез. докл. IV межд. конф. — Нижний Новгород, 1997. — С. 37–47.
(http: // www.rapros97.nnov.ru / reports / 1.html)

[33] **Тей А., Грибомон П., Луи Ж., Снийерс Д., Водон П., Гоше П., Грегуар Э., Санчес Э., Дельсарт Ф.** Логический подход к искусственному интеллекту: от классической логики к логическому программированию. — М.: Мир, 1990. — 432 с.

[34] **Фути К., Судзуки Н.** Языки программирования и схемотехника СБИС. — М.: Мир, 1988. — 224 с.

[35] **Ханссон А., Хариди С., Тернлунд С.-А.** Свойства одного языка логического программирования // Логическое программирование / Под ред. В. Н. Агафонова. — М.: Мир, 1988. — С. 230–244.

[36] **Хоггер К.** Введение в логическое программирование. — М.: Мир, 1988. — 348 с.

[37] **Хьюитт К.** Открытые системы // Реальность и прогнозы искусственного интеллекта. — М.: Мир, 1987. — С. 85–102.

[38] **Хювёнен Э., Сеппянен Й.** Мир Лиспа: В 2 т. — М.: Мир, 1990.

[39] **Чень Ч., Ли Р.** Математическая логика и автоматическое доказательство теорем. — М.: Наука, 1983.

[40] **Чери С., Готлоб Г., Танка Л.** Логическое программирование и базы данных. — М.: Мир, 1992. — 352 с.

[41] **Чэн Ш.-К.** Принципы проектирования систем визуальной информации. — М.: Мир, 1994. — 408 с.

[42] **Шапиро Э., Такеути А.** Объектно-ориентированное программирование в Параллельном Прологе // Язык Пролог в пятом поколении ЭВМ / Под ред. Н. И. Ильинского. — М.: Мир, 1988. — С. 71–102.

[43] **Шлеер С., Меллор С.** Объектно-ориентированный анализ: моделирование мира в состояниях. — Киев: Диалектика, 1993. — 240 с.

[44] **Agha G., Mason I. A., Smith S., Talcott C.** Towards a Theory of Actor Computation (Extended Abstract) // Proc. Third Intern. Conf. on Concurrency Theory (CONCUR'92). — Berlin: Springer-Verlag, 1992. — PP. 565–579.

[45] **Alashqur A. M., Su S. Y. W., Lam H.** Constraint Specification on Object-Oriented Databases // Proc. 1992 Intern. Conf. on Computer Languages. — Los Alamitos: IEEE Computer Society Press, 1992. — PP. 300–309.

[46] **Alexiev V.** Mutable Object State for Object-Oriented Logic Programming: A Survey: Technical report TR93-15 / Dep. of Computing Science, University of Alberta. — Alberta, Canada, 1993. — 23 p.
(ftp: // ftp.cs.ualberta.ca, документ / pub / oolog / state.ps.Z)

[47] **Alf-Christian Achilles** The Collection of Computer Science Bibliographies / University of Karlsruhe. — Karlsruhe, Germany, 1997.
(http: // liinwww.ira.uka.de / bibliography ?)

[48] ALS Prolog Compilers: Object-Oriented Subsystem / Applied Logic Systems, Inc. — Cambridge, USA, 1997.
(http: // www.als.com / als / alspro/ objectpro.html)

[49] **Andreoli J.-M., Leth L., Pareschi R., Thomsen B.** True Concurrency Semantics for a Linear Logic Programming Language with Broadcast Communication // TAPSOFT'93: Theory and Practice of Software Development / Ed.: M. C. Gaudel, J.-P Jouannaud. — Berlin: Springer-Verlag, 1993. — PP. 182–198.

[50] **Benkerimi K., Hill P. M.** Object-oriented Programming in Gödel: An Experiment // Meta-Programming in Logic: Proc. Third Intern. Workshop META-92 / Ed.: A. Pettorossi. — Uppsala, Sweden, 1992. — PP. 177–191.

[51] **Bergmann F., Ostermann M., von Walter G.** Brain Aid Prolog / Technische Universität Berlin. — Berlin, Germany, 1997. (http: // flp.cs.tu-berlin.de / ~fraber / bap.html)

[52] **Berg K., Kalinichenko L.** Modeling Facilities for the Component-based Software Development Method // ADBIS'96: Proc. of the Third Intern. Workshop on Advances in Databases and Information Systems. — Moscow: MEPhI, 1996. — PP. 10–20.

[53] **Bonner A. J., Kifer M., Consens M.** Database Programming in Transaction Logic // Database Programming Languages / Ed.: C. Beeri, A. Ohori, D. E. Shasha. — Berlin: Springer-Verlag, 1994. — PP. 309–337.

[54] CASE.Аналитик для IBM PC. — М.: Эйтэкс, 1993. — 57 с.

[55] **Castelfranchi C., Cesta A., Conte R., Miceli M.** Foundations for Interaction: The Dependence Theory // Advances in Artificial Intelligence / Ed.: P. Toracso. — Berlin: Springer-Verlag, 1993. — PP. 59–64.

[56] **Colmerauer A., Kanoui H., Pasero R., Roussel P.** Un système de communication homme-machine en français: Rapport groupe d'intelligence artificielle / Université d'Aix-Marseille. — Aix-Marseille, France, 1973.

[57] **Diaz M., Pimentel E., Troya J. M.** DROL: A Distributed and Real-time Object-oriented Logic Environment // The Computer Journal. — 1994. — V. 37. — № 5. — PP. 407–421.

[58] **Eliëns A., de Vink E. P.** Asynchronous rendez-vous in distributed logic programming // Semantics: Foundations and Applications / Ed.: J. W. de Bakker, W.-P. de Roever, G. Rozenberg. — Berlin: Springer-Verlag, 1993. — PP. 174–203.

[59] **Embley D. W., Liddle S. W., Ng Y.-K.** Harmonically Combining Active, Object-Oriented and Deductive Databases // ADBIS'96: Proc. of the Third Intern. Workshop on Advances in Databases and Information Systems. — Moscow: MEPhI, 1996. — PP. 21–30.

[60] **Farley J. F., Varhol P. D.** A Visual Approach to Data Acquisition // Dr. Dobb's Journal. — 1993. — V. 18. — № 5. — PP. 145–147.

[61] **Fernandes A. A., Paton N. W., Williams M. H., Bowles A.** Approaches to deductive object-oriented databases // Information and Software Technology. — 1992. — V. 34. — № 12. — PP. 787–803.

[62] **Floyd M.** Roll Your Own Object-Oriented Language // Dr. Dobb's Journal. — 1990. — V. 15. — № 11. — PP. 16–18.

[63] **Giordano L., Martelli A.** A Modal Framework for Structured Logic Programs // Extensions of Logic Programming / Ed.: E. Lamma, P. Mello. — Berlin: Springer-Verlag, 1993. — PP. 168–186.

[64] **Goble T.** Structured Systems Analysis through Prolog — London: Prentice Hall, 1990. — 384 p.

[65] **Hewitt C.** Open Information Systems Semantics for Distributed Artificial Intelligence // Artificial intelligence. — 1991. — V. 47. — № 1/3. — PP. 79–106.

[66] **Hewitt C.** Viewing Control Structures as Patterns of Passing Messages // Artificial intelligence. — 1977. — V. 8. — № 3. — PP. 323–364.

[67] **Horstmann T. C.** A Logical Approach for Distributed Truth Maintenance // Progress in Artificial Intelligence / Ed.: M. Filgueiras, L. Damas. — Berlin: Springer-Verlag, 1993. — PP. 29–44.

[68] **Jacobson I., Christerson M., Jonsson P., Övergaard G.** Object-Oriented Software Engineering. A Use Case Driven Approach. — New York: ACM Press, 1994. — 528 p.

[69] KAPPA-PC Version 2.4. System Description and Data Sheet / IntelliCorp. — Mountain View, USA, 1997.
(http: // www.intellicorp.com / kappa-pc /)

[70] **Kozlov A. V., Kopylov G. V., Obukhov Yu. V.** Graphical user interface of systems for measurements and control // Information Technology. Image Processing and Pattern Recognition: Proc. 1 Intern. Conf. — Lvov. — 1990. — V. 2. — PP. 52–55.

[71] Logic Programming / Oxford University Archive. — Oxford, UK, 1997.
(http: // www.comlab.ox.ac.uk / archive / logic-prog.html)

[72] **Manwu X., Jianfeng L., Fancong Z., Jingwen D.** Agent Language NUML and Its Reduction Implementation Model Based on HOπ // ACM SIGPLAN Notices. — 1994. — V. 29. — № 5. — PP. 41–48.

[73] **Milanese V.** A Proposal for a Distributed Model of GKS Based on Prolog // Computer Graphics Forum. — 1988. — V. 7. — № 3. — PP. 203–213.

[74] **Moura P.** Logtalk 1.5. User Manual (preliminary draft) / Dep. of Mathematics, University of Coimbra. — Coimbra, Portugal, 1997.
(http: // cygnus.ci.uc.pt / logtalk / manual / toc.html)

[75] **Mumick I. S., Ross K. A.** Noodle: A Language for Declarative Querying in an Object-Oriented Database // Deductive and Object-Oriented Databases / Ed.: S. Ceri, T. Tanaka, S. Trur. — Berlin: Springer-Verlag, 1993. — PP. 360–378.

[76] **Obukhov Yu. V.** Visualization in experimental radiophysisc and electronics // Pattern Recognition and Image Analysis: Advanced Mathematical Theory and Application. — 1991. — V. 1. — № 4. — PP. 450–459.

[77] PDC PROLOG Professional User's Guide. — Copenhagen: Prolog Development Centre.
(http: // www.pdc.dk)

[78] **Pimentel E., Troya J. M.** Compositionality Issues of Concurrent Object-Oriented Logic Languages // PARLE'93: Parallel Architectures and Languages Europe / Ed.: A. Bode, M. Reeve, G. Wolf. — Berlin: Springer-Verlag, 1993. — PP. 529–540.

[79] **Pountain D.** Adding Objects to Prolog // Byte. — 1990. — V. 15. — № 8.

[80] **Russell S. J., Norvig P.** Artificial Intelligence. A Modern Approach. — London: Prentice-Hall, 1995. — 932 p.

[81] SICStus Advanced Prolog Technology / Swedish Institute of Computer Science. — Kista, Sweden, 1997.
(http: // www.sics.se / ps / sicstus.html)

Приложение А

Определение Акторного Пролога

(версия от 14 сетября 2001 г.)

А́кторный Пролог — объектно-ориентированный логический язык, реализующий логическую акторную модель информационных систем, разработанную в ИРЭ РАН для исследования и применения методов визуального объектно-ориентированного логического программирования при решении задач структурного и объектно-ориентированного анализа и проектирования, предполагающих:

- проверку и логическое доказательство правильности соединения компонентов проектируемой системы и анализ ее свойств;
- интерактивное проектирование систем, выявление и устранение смысловых противоречий, возникающих в ходе интерактивного построения системы;
- имитационное моделирование, тестирование и объектно-ориентированное логическое программирование систем.

Отправной точкой для разработки логической акторной модели послужила акторная модель вычислений Хьюитта, однако в отличие от нее, логическая акторная модель представляет информационную систему в виде теоремы на некотором логическом языке, разделенной на «логические акторы» — повторно доказываемые подцели, взаимодействующие через общие

переменные; доказательство логических акторов (далее — просто «акторов») осуществляется в объектно-ориентированном пространстве поиска, топология которого соответствует структуре системы. Логическая акторная модель вычислений, соответствующая такому представлению информационной системы, основана на повторных доказательствах акторов, осуществляемых при изменении значений общих переменных, в целях обеспечения корректности и полноты доказательства теоремы. С точки зрения логической акторной модели вычислений, взаимодействие человека и машины рассматривается как доказательство некоторой теоремы, в котором одновременно принимают участие человек, изменяющий исходные данные, и машина, обеспечивающая корректность и полноту доказательства. Основным достоинством логической акторной модели является возможность построения на ее основе логически корректной интерпретации реагирующих систем, необратимых процессов и разрушающего присваивания в логическом языке программирования. В рамках этой модели предполагается, что разрушающее присваивание, изменяющее значения общих переменных программы, вызывает повторное доказательство акторов, зависящих от старых значений переменных, и объявляется успешным в том и только в том случае, если все эти повторные доказательства завершаются успехом, а необратимые процессы рассматриваются как задержка восстановления состояний акторов в случае отката программы. При этом внешние воздействия, вызывающие отклик информационной системы, (в том числе изменение человеком условий задачи во время логического доказательства) интерпретируются как использование разрушающего присваивания, вызывающего повторное доказательство акторов логической программы. В целях реализации логической акторной модели вычислений, в Акторном Прологе разработана логическая интерпретация понятий объектно-ориентированного программирования, отражающая его динамические, структурные и информационные аспекты. Перечисленные аспекты объектно-ориентированного программирования реализуются в Акторном Прологе с помощью:

- акторного механизма;
- классов, миров и наследования;
- простых и составных термов.

Все синтаксические конструкции Акторного Пролога, за исключением четко очерченного набора вспомогательных средств управления, имеют

строгую декларативную семантику и могут быть однозначно представлены в виде формул логики предикатов первого порядка. Различаются «минимальная», «быстрая» и «максимальная» версии языка. Минимальной и быстрой версиям соответствует упрощенный алгоритм унификации (без проверки вхождения). В максимальной и быстрой версиях допускается параллельное исполнение повторных доказательств акторов. В минимальной версии осуществляется полное восстановление предыдущего состояния программы в случае отката. Для определения синтаксиса языка используется расширенная форма Бэкуса-Наура. Терминальные символы, когда это необходимо, выделяются с помощью кавычек и апострофов.

Ссылки на различные части этого документа даются в форме 1.2.3 (приложение **A**, раздел 1.2.3).

Настоящий раздел (введение), а также любые примеры, примечания и ссылки не являются составной частью определения языка.

Ссылки: актор 7.1, акторный механизм 7, доказательство акторов 7.3, задержанный актор 7.3.3, значение переменной 3.1, класс 4.1, мир 4.1, общие переменные 7.2, откат программы 7.3.3, переменная 2.1.1, повторное доказательство 7.1, подцель доказательства 5, проверка вхождения 3.4, программа 4, простой терм 3.1, разрушающее присваивание 7.4, составной терм 3.2, средства управления 4, унификация термов 3.4.

A.1 Алфавит языка

В качестве алфавита языка используется набор символов ASCII, при этом различаются графические символы (графемы), имеющие визуальное представление в виде отпечатанного знака или пробела, и управляющие символы: возврат на одну позицию, горизонтальная табуляция, перевод строки, вертикальная табуляция, перевод формата и возврат каретки. Минимальный набор графических символов, достаточный для определения языка, включает буквы, цифры, символ пробела и специальные символы.

буква = большая_буква | маленькая_буква
большая_буква =
"A" | "B" | "C" | "D" | "E" | "F" | "G" |
"H" | "I" | "J" | "K" | "L" | "M" | "N" |
"O" | "P" | "Q" | "R" | "S" | "T" | "U" |
"V" | "W" | "X" | "Y" | "Z"
маленькая_буква =
"a" | "b" | "c" | "d" | "e" | "f" | "g" |

“h” | “i” | “j” | “k” | “l” | “m” | “n” |
“o” | “p” | “q” | “r” | “s” | “t” | “u” |
“v” | “w” | “x” | “y” | “z”

цифра =
“0” | “1” | “2” | “3” | “4” | “5” | “6” |
“7” | “8” | “9”

буквы_и_цифры =
[буквы_и_цифры [“_”]] буква_или_цифра

буква_или_цифра = буква | цифра

К специальным символам относятся:
! " # & ' ( ) * + , − . / : ; < = > ? @ [\ ] _ ‘ { | }

A.2 Лексика

Текст программы рассматривается как последовательность лексем и разделителей. Разделителями являются комментарии, а также пробелы и управляющие символы, не входящие в состав лексем и комментариев. Чтобы обеспечить однозначность трансляции текста, приняты следующие соглашения:

1. Сканирование текста всегда осуществляется слева направо.
2. В состав каждой лексемы включается по возможности большее число графических символов.
3. Фрагмент текста «:–» не является лексемой, если он расположен между лексемами «{» и «}».

Пример. Последовательность лексем и разделителей.

Текст «P{a:−7}:–P{/∗/b:0}.−−1−−» содержит лексемы «P», «{«, «a», «:», «−», «7», «}», «:–», «P», «{«, «b», «:», «0», «}», «.» и комментарии «/∗/», «−−1−−».

Ссылки: графема 1, комментарий 2.2, лексема 2.1, программа 4, управляющий символ 1.

A.2.1 Лексемы

Лексемами являются: переменные, символы и ключевые слова, целые числовые литералы, вещественные числовые литералы, сегменты строк, ограничители. В ходе сканирования текста происходит преобразование информации, поэтому в определении языка различаются собственно «лексемы», воспринимаемые лексическим анализатором, и «значения лексем», которые обрабатывает синтаксический анализатор.

Ссылки: ключевое слово 2.1.2, ограничитель 2.1.5, переменная 2.1.1, сегмент строки 2.1.4, символ 2.1.2, числовой литерал 2.1.3.

A.2.1.1 Переменные

Переменная — это имя, начинающееся с большой буквы или символа подчеркивания «_».

переменная =
    большая_буква [["_"] буквы_и_цифры] |
    "_" [буквы_и_цифры]

Маленькие буквы в составе переменной заменяются соответствующими большими буквами, при этом все остальные графемы остаются без изменений. Полученная последовательность графем считается значением лексемы.

Переменная «_» называется «анонимной». Считается, что все анонимные переменные в тексте программы являются некоторыми уникальными, однократно использованными именами.

Пример. Правильно построенные переменные:
A1, _, AbC_Ef_H7, _7, Variable, _X_123

Ссылки: большая буква 1, буквы и цифры 1, графема 1, значение лексемы 2.1, маленькая буква 1, программа 4.

A.2.1.2 Символы и ключевые слова

Символ — это имя, начинающееся с маленькой буквы или заключенное в апострофы:

символ =
    маленькая_буква [["_"] буквы_и_цифры] |
    ' { графема } '

Большие буквы в составе символа заменяются соответствующими маленькими буквами, при этом все остальные графемы остаются без изменений. Полученная последовательность графем считается значением символа. Апострофы, в которые может быть заключен символ, не являются составными частями его значения. Если апострофы не используются, значение символа не может совпадать с ключевыми словами языка.

Ключевыми словами являются следующие имена:

and	— и	**class**	— класс
div	— деление	**external**	— внешний
if	— если	**is**	— является
mod	— по_модулю	**or**	— или
project	— проект	**using**	— использующий

Для написания ключевых слов языка используются только маленькие буквы. Значениями ключевых слов считаются соответствующие цепочки графем.

Пример. Правильно построенные символы:

symbol, 'ALPHA', abc_EF_h, '', s4734

Ссылки: большая буква 1, буквы и цифры 1, графема 1, значение лексемы 2.1, маленькая буква 1.

A.2.1.3 Числовые литералы

Числовой литерал — это лексема, обозначающая числовое значение:

числовой_литерал =
    цифры ["." цифры] [порядок] |
    цифры "#" буквы_и_цифры "#" [порядок] |
    ' графема

Числовые литералы бывают целые и вещественные (плавающие) — значениями таких литералов являются, соответственно, (беззнаковые) целые и вещественные числа.

По умолчанию основание числового литерала равно 10, в целых числовых литералах основание может быть указано явно. Основание и порядок числовых литералов всегда записываются в десятичной системе. В качестве (расширенных) цифр от 10 до 35 используются латинские буквы от «A» до «Z» (от «a» до «z») соответственно. Значение каждой (расширенной) цифры литерала с основанием должно быть меньше основания.

Вещественные числа соответствуют числовым литералам, содержащим точку. В языке не гарантируется точное представление вещественных чисел, в мантиссе которых количество значащих цифр превышает значение, соответствующее максимальной относительной погрешности D, определяемой конкретной реализацией языка. В качестве значений таких числовых литералов принимаются некоторые близкие числа, отличающиеся от них на величину, не превышающую D.

Если в качестве числового литерала используется последовательность ‘ графема, его значением является числовой код заданного графического символа (целое число).

цифры = [цифры [“_”]] цифра

Символы подчеркивания между соседними цифрами и буквами числового литерала не влияют на его значение.

порядок = буква_e [“+” | “−”] цифры
буква_e = “E” | “e”

Для получения значения числового литерала с порядком необходимо умножить значение числового литерала без порядка на основание, возведенное в указанную порядком степень. Порядок целых числовых литералов не может содержать знак минус.

Пример. Правильно построенные числовые литералы:
13_274, 2#1100_0100#E4, 39.123e100, 8#177_777#,
3.217_514e+90, 16#EF93#, ‘y, 3.51E−31

Ссылки: буква 1, буквы и цифры 1, графема 1, значение лексемы 2.1, лексема 2.1, цифра 1.

A.2.1.4 Сегменты строк

Сегмент строки — это лексема, обозначающая цепочку графических и управляющих символов:

сегмент_строки = ‘"’ { графема | “\” код } ‘"’

В ходе сканирования сегмента строки конструкции вида «\»код (где код — некоторая буква или числовой литерал) заменяются соответствующими графическими и управляющими символами.

код = “b” | “t” | “n” | “v” | “f” | “r” | числовой_литерал

Буквенные коды соответствуют управляющим символам:

b — возврат на одну позицию;
n — перевод строки;
f — перевод формата;
t — горизонтальная табуляция;
v — вертикальная табуляция;
r — возврат каретки.

В качестве кода в сегменте строки не допускается (считается синтаксической ошибкой) использование вещественных числовых литералов, а также числовых литералов, значения которых лежат за пределами некоторого интервала, определяемого конкретной реализацией языка. В случае если графический символ, следующий после «\», не является кодом, переключатель «\» игнорируется, а обнаруженный за ним графический символ включается в сегмент строки без дальнейшего анализа. Полученная таким образом последовательность графических и управляющих символов (не считая кавычек, в которые заключен сегмент строки) является значением сегмента строки.

Пример. Правильно построенные сегменты строк:
"String \"XYZ\"\n", "", "c:\\dos*.*"

Ссылки: буква 1, графема 1, значение лексемы 2.1, лексема 2.1, управляющий символ 1, числовой литерал 2.1.3.

A.2.1.5 Ограничители

Ограничитель — это последовательность из одного или нескольких специальных символов, используемая в синтаксических конструкциях языка. В языке используются:

1. простые ограничители
   ! & ( ) * + , − . / : ; < = > ? @ [ ] { | }
2. составные ограничители
   :− == := <> <= >=

Значениями ограничителей считаются соответствующие цепочки графем.

Ссылки: графема 1, значение лексемы 2.1, специальный символ 1.

A.2.2 Комментарии

Комментарием является последовательность графических и управляющих символов, начинающаяся с открывающей скобки комментария и заканчивающаяся закрывающей скобкой; комментариям разных типов соответствуют

разные скобки. Открывающая скобка не является началом комментария, если ее графические символы входят в состав лексемы или другого комментария. Определены два типа комментариев:

1. Однострочный комментарий (открывающая скобка — два соседних дефиса; закрывающая — любой управляющий символ, кроме горизонтальной табуляции).

2. Многострочный комментарий (открывающая и закрывающая скобки «/*» и «*/» соответственно; повторное вхождение открывающей скобки «/*» в состав многострочного комментария считается синтаксической ошибкой).

Пример. Правильно построенные комментарии:

```
-- Однострочный комментарий

                /*/
/* Многострочные комментарии */
                /*/
```

Ссылки: графема 1, лексема 2.1, управляющий символ 1.

A.3 Определение данных

Для определения данных в языке используются термы и выражения.

терм = простой_терм | составной_терм | вызов_функции

Значениями термов и выражений (всегда) являются элементы данных или — если речь идет о несвязанных переменных — значения лексем «переменные». Элементы данных создаются в ходе исполнения вызовов предикатов, во время формирования слотов миров, а также во время глобальных операций с общими переменными. В дальнейшем, когда будет идти речь об унификации и других операциях с термами, следует иметь в виду обработку значений термов.

Вызовы функций в различных синтаксических конструкциях разрешается использовать только в составе собственных предложений.

Ссылки: вызов функции 5.1.2, выражение 3.3, глобальные операции 7.2, значение лексемы 2.1, исполнение предиката 5, мир 4.1, несвязанная переменная 3.1, переменная 2.1.1, простой терм 3.1, собственное предложение 5.1, составной терм 3.2, унификация термов 3.4, формирование слота 4.1.4.

A.3.1 Простые термы

Простой терм — это элементарная синтаксическая конструкция, обозначающая данные. В качестве простых термов используются переменные, символы, целые числа, вещественные числа, строковые литералы:

простой_терм = константа | переменная
константа =
    символ | ["–"] числовой_литерал | строковый_литерал

Строковый литерал — это последовательность сегментов строки, обозначающая цепочку графических и управляющих символов:

строковый_литерал = [строковый_литерал] сегмент_строки

Считается, что значением терма «переменная» является значение лексемы «переменная», до тех пор пока переменная (терм) не будет связана с каким-либо элементом данных — константой или составным термом. Значением связанной переменной считается соответствующий элемент данных. Переменная, не связанная с элементом данных, называется «несвязанной».

Значения других простых термов определяются значениями соответствующих им лексем.

Диапазоны допустимых целых и вещественных чисел определяются конкретной реализацией языка, при этом значения целых числовых литералов с явно указанным основанием (выходящие за пределы допустимого диапазона) разрешается рассматривать в качестве битового представления отрицательных чисел.

Значением строкового литерала является конкатенация значений последовательности входящих в его состав сегментов строк. Максимальная допустимая длина значения строкового литерала определяется конкретной реализацией языка.

Пример. Правильно построенные простые термы:
VARIABLE, symbol, −2#0100_1100#, 34.0e−9, "STRING" "_" "OF" "_" "TEXT"

Ссылки: графема 1, данные 3, значение лексемы 2.1, значение терма 3, лексема 2.1, переменная 2.1.1, сегмент строки 2.1.4, символ 2.1.2, составной терм 3.2, терм 3, управляющий символ 1, числовой литерал 2.1.3.

A.3.2 Составные термы

Составными термами являются структуры, списки и недоопределенные множества:

составной_терм =
    структура | список | недоопределенное_множество

Значения составных термов определяются с помощью кортежей и некоторых специальных имен (констант).

Ссылки: значение терма 3, недоопределенное множество 3.2.3, список 3.2.2, структура 3.2.1, терм 3.

A.3.2.1 Структуры

Структура — это составной терм, построенный из функтора и последовательности одного или более аргументов, заключенной в круглые скобки:

структура = символ "(" выражения ")"

Значением структуры $f(A_1,A_2,\ldots,A_n)$ является кортеж длины $n+2$, в первой позиции которого стоит специальная константа #structure:

$<$#structure,f,A_1,A_2,. . . ,$A_n>$.

Пример. Правильно построенные структуры:
g1(1+2,X,Y), functor(i(1−(R∗12),2,3),4,k(5),Z), h(J)

Ссылки: выражения 3.3, значение терма 3, символ 2.1.2, составной терм 3.2.

A.3.2.2 Списки

Список — это составной терм, построенный из последовательности (возможно, пустой) аргументов, заключенной в квадратные скобки. В случае если последовательность аргументов списка не является пустой, в его состав может быть включен дополнительный компонент — переменная, обозначающая остаток (хвост) списка:

список = "[" [выражения ["|" хвост]] "]"
хвост = переменная | вызов_функции

Значением пустого списка [ ] является специальная константа

#empty_list.

Значением списка [A_1,A_2,...,A_n|Rest] является кортеж

<#list,A_1,<#list,A_2,...<#list,A_n,Rest>...>>,

где #list — специальная константа, Rest — хвост списка.

Таким образом, терму [A_1,A_2,...,A_n] соответствует значение

<#list,A_1,<#list,A_2,...<#list,A_n,#empty_list>...>>.

Пример. Правильно построенные списки:
[17,_,"item_of_list",321,93,_], [X+721,Y,R+H,Z|R], [ ]

Ссылки: вызов функции 5.1.2, выражения 3.3, значение терма 3, переменная 2.1.1, составной терм 3.2, терм 3.

A.3.2.3 Недоопределенные множества

Недоопределенное множество — это составной терм, построенный из набора (возможно, пустого) элементов, заключенного в фигурные скобки. Элементы недоопределенного множества задаются в виде пар

«имя_элемента: значение_элемента»,

где имя элемента — некоторый символ или неотрицательное целое число, а значение элемента — произвольное выражение:

элементы_множества =
[элементы_множества “,”] элемент_множества
элемент_множества = имя_элемента [“:” выражение]
имя_элемента = символ | числовой_литерал

В случае если значение некоторого элемента недоопределенного множества не задано, значением такого элемента считается анонимная переменная «_».

Ключом недоопределенного множества называется значение элемента с именем '' (символ '' состоит из пустой цепочки графем), которое может быть задано в начале недоопределенного множества, за пределами фигурных скобок. При таком способе определения в качестве ключа недоопределенного множества разрешается использовать только простые термы и атрибуты:

ключ = простой_терм | атрибут

Недоопределенное множество вида

$F\{x_1{:}A_1,x_2{:}A_2,\ldots,x_n{:}A_n|Rest\}$,

в составе которого задан ключ F, эквивалентно

$\{''{:}F,x_1{:}A_1,x_2{:}A_2,\ldots,x_n{:}A_n|Rest\}$.

В случае если набор элементов множества (учитывая ключ) не является пустым, в составе множества может быть задан дополнительный компонент — переменная, обозначающая неопределенный остаток (хвост) множества.

недоопределенное_множество =
[ключ] "{" [элементы_множества] ["|" хвост] "}"

Недоопределенное множество не может содержать пары с одинаковыми именами элементов.

Для того чтобы построить значение недоопределенного множества, необходимо:

1. Просмотреть полный текст программы и построить множество S всех имен элементов всех недоопределенных множеств, которые в ней используются.

2. С помощью лексикографического упорядочения из элементов множества S построить цепочку $s_1,s_2,\ldots,s_m$ (m — мощность множества S).

3. Если недоопределенное множество обозначает конечное число элементов (случай $\{s_x{:}A_x,s_y{:}A_y,\ldots,s_z{:}A_z\}$), его значением является кортеж длины m+1:

   $<\#set,\ldots,\#,\ldots,t(A_y),\ldots,\#,\ldots,t(A_x),\ldots,t(A_z),\ldots>$,

   где #set — специальная константа, t — имя вспомогательного функтора. Каждая позиция i+1 (i=1,...,m) кортежа содержит значение $t(A_i)$, если пара с именем элемента s_i присутствует в рассматриваемом терме, или, если такая пара не обнаружена, специальную константу #. Значением пустого множества $\{\}$, в частности, является кортеж вида

   $<\#set,\#,\#,\#,\#,\ldots\#,\#,\#>$.

4. Значением недоопределенного множества общего вида

   $\{s_x{:}A_x,s_y{:}A_y,\ldots,s_z{:}A_z|Rest\}$

   является кортеж длины m+1:

$$G_1 = <\#set,\ldots,V_1,\ldots,t(A_y),\ldots,V_2,\ldots,t(A_x),\ldots,t(A_z),\ldots>,$$

каждая позиция i+1 (i=1,...,m) которого содержит $t(A_i)$, если пара с именем элемента s_i присутствует в рассматриваемом терме, или некоторую уникальную переменную V_j, если соответствующая пара не обнаружена. Кроме того, понадобится еще один кортеж

$$G_2 = <\#set,\ldots,V_1,\ldots,\#,\ldots,V_2,\ldots,\#,\ldots,\#,\ldots>,$$

который отличается от предыдущего тем, что все аргументы t(A) в нем заменены константами #. Считается, что всякий раз, когда создается значение недоопределенного множества G_1, одновременно с этим происходит унификация G_2 с переменной Rest.

Пример. Правильно построенные недоопределенные множества:
{x:17,y:−(1+X),z,5:'x'}, R2{q:Y,e:W,symbol:_,k:0|W}, {}

Ссылки: анонимная переменная 2.1.1, атрибут 4.1.1, выражение 3.3, графема 1, значение терма 3, переменная 2.1.1, программа 4, простой терм 3.1, символ 2.1.2, составной терм 3.2, терм 3, унификация термов 3.4, хвост 3.2.2, числовой литерал 2.1.3, '' 2.1.2.

A.3.3 Выражения

Выражение — это атрибут или видоизмененный терм:

выражение = [выражение аддитивный_оператор] слагаемое
слагаемое =
[слагаемое мультипликативный_оператор] множитель
множитель = терм | атрибут | ["−"] "(" выражение ")"
выражения = [выражения ","] выражение

Для построения выражений используется ограниченный набор знаков операций, в состав которого входят математические символы:

аддитивный_оператор = "+" | "−"
мультипликативный_оператор = "∗" | "/" | **div** | **mod**

Выражение, построенное с помощью инфиксного знака операции, эквивалентно структуре вида

символ(аргумент$_1$,аргумент$_2$),

где символ — знак операции, заключенный в апострофы, аргумент$_1$ и аргумент$_2$ — операнды, стоящие слева и справа от знака операции.

Выражение, построенное с помощью префиксного знака операции «−», эквивалентно структуре вида

'−'(аргумент),

аргументом которой является операнд выражения.

Пример. Правильно построенные выражения:
1+H∗−(E)/4**div**(W+"A4"−319e0), −7, f∗X+(7**mod**"t")−r

Ссылки: атрибут 4.1.1, символ 2.1.2, структура 3.2.1, терм 3, **div** 2.1.2, **mod** 2.1.2.

A.3.4 Унификация термов

Унификация термов осуществляется в ходе исполнения вызовов предикатов, в момент создания значений недоопределенных множеств, а также во время глобальных операций с общими переменными. Кроме того, унификация термов может быть вызвана явно с помощью встроенного предиката

L == R.

Встроенный предикат унификации разрешается использовать с произвольным количеством аргументов:

'=='(V$_1$,...,V$_k$).

Константы и составные термы разных видов несопоставимы между собой (унификация таких термов невозможна). В максимальной версии языка в ходе унификации осуществляется проверка вхождения, в быстрой версии проверка вхождения используется только во время глобальных операций с общими переменными, в минимальной версии языка проверка вхождения не используется, однако во время глобальных операций, при обнаружении переменной, связанной с термом, содержащим эту переменную, допускается вызов исключительных ситуаций.

Пример. Унификация двух составных термов:

{region:X,name:"Baikal"|Rest1}
==
{name:Y,object:lake,region:"Siberia"}

В ходе унификации элементы одного множества будут унифицированы с элементами другого в соответствии с заданными именами элементов. Результатом унификации станут подстановки X="Siberia", Y="Baikal", на месте переменной Rest1 окажется значение G недоопределенного множества, включающего одну-единственную пару object:lake. Остальные позиции кортежа G (кроме первой, которая всегда содержит #set) будут заполнены #.

Ссылки: встроенный предикат 8, глобальные операции 7.2, значение терма 3, исключительная ситуация 7.7, исполнение предиката 5, константа 3.1, недоопределенное множество 3.2.3, переменная 2.1.1, связывание переменной 3.1, составной терм 3.2, терм 3, элемент множества 3.2.3.

A.4 Структура программы

Программа состоит из множества классов и целевого утверждения («проекта»):

программа = { определение_класса | определение_проекта }

Будем говорить, что некоторый класс C (или проект) «использует» класс E, если E является предком C в иерархии наследования классов, а также если C (проект) или кто-то из его предков содержит конструктор экземпляра класса E (или конструктор экземпляра класса F, такого что класс F использует класс E), не считая тех конструкторов, которые содержатся в классах, используемых классом C (проектом), только в качестве инициализаторов, перекрываемых во время построения соответствующих миров. В программе должны быть определены все классы, используемые проектом.

Средствами управления называются синтаксические средства, не имеющие декларативной семантики: встроенные управляющие операторы, заголовки внешних предложений и акторный префикс «&». Декларативная семантика программы, в которой не используются средства управления, эквивалентна декларативной семантике соответствующей программы без акторных префиксов. Такая программа может быть однозначно представлена в виде формулы логики предикатов первого порядка.

Исполнением программы называется доказательство существования согласованного состояния набора акторов, возникающих во время этого доказательства. Исполнение программы начинается с доказательства ее проекта.

Ссылки: актор 7.1, акторный префикс 5.1.1, встроенный оператор 8, заголовок внешнего предложения 5.2, иерархия наследования 4.1, инициализатор 4.1.2, конструктор 4.1.3, мир 4.1, определение класса 4.1, определение проекта 4.2, перекрытие инициализаторов 4.1.4, построение миров 4.1.4, согласованное состояние 7.2.

A.4.1 Классы

Класс — это набор предложений языка, имеющий уникальное имя и входящий в состав иерархии наследования:

определение_класса =
    class заголовок_класса **is** атрибуты "[" предложения "]"

В языке используется одиночное наследование: у класса может быть не более одного непосредственного предка и неограниченное число потомков. Имя непосредственного предка указывается в определении после имени класса:

заголовок_класса = имя_класса [**using** имя_класса]
имя_класса = символ

В иерархии наследования классов, используемых проектом, запрещены циклические зависимости.

Экземплярами классов («мирами») называются конкретные применения классов; они являются составными частями пространства поиска в ходе исполнения программы.

Экземпляр класса характеризуется набором предложений соответствующего класса и его предков, а также набором слотов, доступных во всех этих предложениях. Построение экземпляров классов происходит в результате доказательства утверждений об их существовании («конструкторов»).

Слот — это составная часть экземпляра класса, характеризуемая именем и значением. Имена слотов называются атрибутами классов. Значением слота может быть либо значение терма, либо мир.

Пример. Правильно построенный класс.

```
class 'MY_WINDOW' using 'WINDOW' is
color = 47
[
goal:-
     shift,!.
]
```

Ссылки: атрибуты 4.1.1, значение терма 3, исполнение программы 4, использование класса 4, конструктор 4.1.3, построение миров 4.1.4, предложения 5, проект 4.2, символ 2.1.2, **class** 2.1.2, goal 4.1.4, **is** 2.1.2, **using** 2.1.2, '!' 8.

A.4.1.1 Атрибуты классов

Имя слота («атрибут») должно быть объявлено во всех классах, в которых используется соответствующий слот. Область действия атрибута распространяется на инициализаторы в определении класса, а также на все предложения класса.

атрибуты = { атрибут ["=" инициализатор] }
атрибут = символ

В составе инициализаторов слотов могут использоваться переменные. Область действия таких переменных ограничена множеством инициализаторов слотов в определении атрибутов класса. В определении атрибутов класса не допускается однократное использование переменных, отличных от «_».

Атрибут self — предопределенный, он обозначает непосредственно тот экземпляр класса, в котором это имя используется.

Повторное определение атрибутов класса (в том числе переопределение атрибута self) считается синтаксической ошибкой. В инициализаторах и предложениях различных классов, используемых проектом и связанных отношением наследования, одни и те же атрибуты должны использоваться одинаково — либо для обозначения данных, либо для обозначения миров.

Пример. Правильно определенные атрибуты класса:

a = Y	e = ('Q',x='+'(a,f),m=self,k=e)
b	f = '∗'(Y,7)
c = f(_,[3,7],Y,a)	d = [ ]
g = _	h = {x:1,y:Y,z:R|_}
i = [0,_,j|R]	j = a

Ссылки: данные 3, иерархия наследования 4.1, инициализатор 4.1.2, использование класса 4, класс 4.1, мир 4.1, переменная 2.1.1, предложения 5, проект 4.2, символ 2.1.2, слот 4.1.

A.4.1.2 Инициализаторы

Инициализатором называется синтаксическая конструкция, определяющая начальное значение слота:

инициализатор = терм | атрибут | конструктор

Значением инициализатора является значение терма или мир.

Ссылки: атрибут 4.1.1, значение терма 3, конструктор 4.1.3, мир 4.1, начальное значение слота 4.1.4, терм 3.

A.4.1.3 Конструкторы

Конструктор — это утверждение о существовании экземпляра класса. Аргументы конструктора определяют значения слотов экземпляра класса.

конструктор =
"(" имя_класса { "," атрибут ["=" инициализатор] } ")"

Значения инициализаторов аргументов конструктора (если только этот конструктор сам не является инициализатором слота, перекрываемым в ходе построения соответствующих миров) должны соответствовать тому, как имена аргументов конструктора — атрибуты — используются в соответствующих классах, используемых проектом.

Отсутствие инициализатора в определении некоторого аргумента конструктора с именем Name является допустимым только в том случае, если рассматриваемый конструктор находится в области действия слота с именем Name. Такое определение аргумента конструктора эквивалентно определению вида «Name=Name».

В конструкторе не допускается определение нескольких аргументов с одинаковыми именами. Не допускается также использование в качестве имени аргумента символа self.

Пример. Правильно построенные конструкторы:
('R53',a=1,b='/'(F,7),c=_,d), ('E'), ('W',q=[_,3,8])

Ссылки: атрибут 4.1.1, значение инициализатора 4.1.2, значение слота 4.1, имя класса 4.1, инициализатор 4.1.2, использование класса 4, класс 4.1, мир 4.1, перекрытие инициализаторов 4.1.4, построение миров 4.1.4, проект 4.2, символ 2.1.2, слот 4.1, self 4.1.1.

A.4.1.4 Построение экземпляров классов

Доказательство конструктора (построение экземпляра некоторого класса C) включает следующие этапы:

1. Формирование экземпляра класса. На этом этапе осуществляется:

   (a) Построение соответствующего пространства поиска.
   В состав пространства поиска входят предложения самого класса C, затем (в соответствии с иерархией наследования) предложения его непосредственного предка D, предложения непосредственного предка класса D и так далее.

   (b) Формирование слотов экземпляра класса.
   Одновременно с созданием каждого слота, если для него задан соответствующий инициализатор, формируется его «начальное» значение — значение терма, если инициализатором слота является терм; некоторый мир, если инициализатором является конструктор; копия значения другого слота, если инициализатором является атрибут, и значение слота, соответствующего этому атрибуту, уже сформировано (в случае если названное значение еще не сформировано, копирование откладывается до того момента, пока это значение не появится).
   В качестве инициализатора каждого формируемого слота используется значение соответствующего аргумента конструктора или, если аргумент не задан, инициализатор в определении класса C или его предка. При этом считается, что инициализаторы, заданные в определении класса C (или его предка E), отменяют («перекрывают») все инициализаторы соответствующих атрибутов в определениях предков класса C (предков класса E).

Слоты, не имеющие инициализаторов, получают в качестве начальных значений уникальные общие переменные.

Все переменные, создаваемые в составе значений слотов, также являются общими.

2. Доказательство предиката goal() во всех мирах, сформированных в ходе построения экземпляра класса C — доказательство конструктора будет считаться успешным, если во всех этих мирах доказательство предиката goal завершится успехом. Порядок доказательств предиката goal в названных мирах выбирается произвольным образом.

(Взаимно-) рекурсивное использование классов, приводящее на этапе формирования слотов к неограниченному увеличению количества миров, считается синтаксической ошибкой.

Примечание. Доказываемые предикаты goal могут быть недетерминированными — в таком случае конструктор экземпляра класса C также будет недетерминированным.

Ссылки: атрибут 4.1.1, значение слота 4.1, значение терма 3, иерархия наследования 4.1, инициализатор 4.1.2, использование класса 4, класс 4.1, конструктор 4.1.3, мир 4.1, общие переменные 7.2, переменная 2.1.1, предложения 5, слот 4.1, терм 3.

A.4.2 Проект

Проектом называется целевое утверждение программы — некоторый конструктор экземпляра класса:

определение_проекта = **project is** конструктор

В составе проекта могут использоваться переменные. Область действия таких переменных ограничена пределами проекта. В определении проекта не допускается однократное использование переменных, отличных от «_».

В качестве обозначения экземпляра класса, соответствующего проекту, в определении проекта допускается использование предопределенного атрибута self.

В ходе исполнения проекта доказательство предикатов goal, осуществляемое на втором этапе исполнения конструктора, рассматривается как доказательство некоторого «корневого» актора программы в мире, соответствующем исполняемому конструктору.

Пример. Правильно построенный проект.
project is ('P',x=[1,'∗',9|W],y=W,p=self)

Ссылки: актор 7.1, атрибут 4.1.1, доказательство акторов 7.3, конструктор 4.1.3, мир 4.1, переменная 2.1.1, программа 4, goal 4.1.4, **is** 2.1.2, **project** 2.1.2, self 4.1.1.

A.4.3 Трансляция программных модулей

Программа может состоять из нескольких программных модулей (исходных файлов), каждый из которых транслируется отдельно от остальных. Определения классов и определение проекта являются «элементарными программными модулями», из которых строится исходный файл. Результат их трансляции — добавление или замена соответствующих «библиотечных модулей» в некоторой программной библиотеке, структура которой должна быть определена в конкретной реализации языка.

Программный модуль не может содержать повторные определения классов. Допускается только одно определение проекта в программном модуле. Максимальная допустимая длина программного модуля определяется конкретной реализацией языка.

Формированием программы называется сборка программы из библиотечных модулей, сопровождаемая проверкой их синтаксической правильности. Формирование программы может осуществляться перед началом или во время ее исполнения, однако синтаксически правильной считается только такая программа, которая может быть сформирована перед началом исполнения.

Примечание. Некоторые синтаксические ошибки обнаруживаются лишь на этапе формирования программы из библиотечных модулей. Такими ошибками являются отсутствие определения проекта или некоторых определений классов в программной библиотеке, циклы в иерархии наследования, несогласованное использование атрибутов в различных классах, связанных отношением наследования, и аргументах конструкторов, неправильное (взаимно-) рекурсивное использование классов в инициализаторах соответствующих им слотов, а также неправильное определение (взаимно-) рекурсивных предикатов переменной арности.

Пример. Исходный файл программы.

```
class 'HELLO' using 'WINDOW' is          -- Определение класса
dx   = 25                                -- Атрибуты
dy   = 10
[                                        -- Предложения
goal:-
     write("Hello world !").
]

project is                               -- Определение проекта
     ('HELLO',x=1,y=2,dx=30,dy=7)
```

Ссылки: атрибут 4.1.1, иерархия наследования 4.1, инициализатор 4.1.2, исполнение программы 4, использование класса 4, конструктор 4.1.3, определение класса 4.1, определение проекта 4.2, предикат переменной арности 6.1, предложения 5, программа 4, слот 4.1, **class** 2.1.2, goal 4.1.4, **is** 2.1.2, **project** 2.1.2, **using** 2.1.2.

A.5 Предложения классов

Различаются «собственные» предложения, «заголовки внешних предложений» и «объявления внешних вызовов».

```
предложение =
     собственное_предложение |
     заголовок_внешнего_предложения |
     объявление_внешних_вызовов
```

В предложении могут использоваться переменные и атрибуты. Область действия переменных ограничена пределами предложения. В собственном предложении не допускается однократное использование переменных, отличных от «_». В составе заголовка внешнего предложения и объявления внешних вызовов не допускается многократное использование переменных, отличных от «_».

Предложения, в заголовке которых используется предикат с переменным числом аргументов, называются метапредложениями.

Предложения класса группируются в соответствии с их заголовками.

предложения = { предложение }

Предложения должны принадлежать одной группе («процедуре»), если совпадают имена и арность предикатных символов заголовков этих предложений. Процедуры, в свою очередь, также должны быть сгруппированы в соответствии с именами предикатных символов заголовков входящих в них предложений.

Считается, что метапредложения не входят в состав каких-либо процедур, однако в определении класса они должны быть сгруппированы с предложениями с такими же именами предикатных символов заголовков.

Исполнение предложений и вызовов предикатов во время исполнения программы осуществляется согласно стандартной стратегии управления, соответствующей текстуальному упорядочению процедур и вызовов (используется так называемый «поиск слева направо в глубину с возвратом»).

Исполнение вызова предиката (исполнение предиката) включает:

1. Выбор предложения (собственного или внешнего), имя заголовка которого совпадает с именем вызываемого предиката.

2. Если следом за выбранным предложением в рассматриваемом мире расположены еще не исследованные предложения, создается новая точка выбора, обозначающая поиск иных предложений, соответствующих условиям пункта 1.

3. Построение копии предложения, отличающейся от выбранного предложения тем, что все его переменные заменяются новыми уникальными именами, а все атрибуты заменяются значениями слотов соответствующего мира, которому принадлежит предложение.

4. Исполнение построенного предложения.

Подцели копии собственного предложения, построенной во время исполнения предиката, называются «подцелями доказательства».

Исполнение предложения включает унификацию аргументов исполняемого вызова предиката с аргументами заголовка предложения, а также:

- для собственного предложения — исполнение соответствующих подцелей доказательства;

- для внешнего предложения — исполнение внешней подпрограммы.

Исполнение предложения заканчивается успехом тогда и только тогда, когда успехом заканчиваются все названные операции.

Пример. Правильно построенная процедура.

```
f(_,_,T,F) ?
f(_,5,x,R):- write("R=",R),!.
f(X,Y,Z,L) [external!]
```

Ссылки: атрибут 4.1.1, вызов предиката 5.1.1, заголовок внешнего предложения 5.2, значение слота 4.1, исполнение программы 4, класс 4.1, мир 4.1, объявление внешних вызовов 5.3, переменная 2.1.1, подцель предложения 5.1.1, предикат переменной арности 6.1, символ 2.1.2, слот 4.1, собственное предложение 5.1, унификация термов 3.4, **external** 2.1.2, '!' 8.

A.5.1 Собственные предложения

Собственное предложение — это логическое правило, состоящее из заголовка и последовательности (возможно, пустой) подцелей. В конце собственного предложения ставится точка.

собственное_предложение = атом [если дизъюнкция] "."
если = ":–" | **if**
дизъюнкция = [дизъюнкция или] конъюнкция
или = **or** | ";"
конъюнкция = [конъюнкция и] подцель
и = "," | **and**

Собственное предложение, в составе которого нет подцелей, называется «фактом».

Ссылки: атом 6, заголовок предложения 5, подцель 5.1.1, подцель предложения 5.1.1, предложение 5, **and** 2.1.2, **if** 2.1.2, **or** 2.1.2.

A.5.1.1 Подцели предложений

Подцелями предложения служат дальние и ближние вызовы предикатов. «Вызовом предиката» называется синтаксическая конструкция, определяющая экземпляр класса, в котором этот вызов должен быть исполнен, тип вызова (ближний или дальний, простой или акторный), а также атомарную формулу вызова. Вызов предиката называется «дальним», если в подцели

явным образом (с помощью атрибута или конструктора) указан мир, в котором он должен быть исполнен. Если соответствующий мир не указан, вызов предиката называется «ближним» и осуществляется в том же самом мире, в котором исполняется рассматриваемое предложение.

Если для обозначения вызова предиката используется префикс «?» (или префикс вообще отсутствует), такой вызов называется «простым». Если используется «акторный» префикс «@» или «&», вызов предиката называется «акторным».

подцель = простая_подцель | бинарное_отношение | "!"
простая_подцель =
[[конструктор_или_атрибут] префикс] простой_атом
конструктор_или_атрибут = конструктор | атрибут
префикс = "?" | "@" | "&"

Подцель «!» обозначает встроенный оператор отсечения '!'.

Если обозначением мира в дальнем вызове предиката является конструктор, доказательство этого конструктора происходит непосредственно перед исполнением предиката, всякий раз, когда осуществляется доказательство подцели.

Ссылки: атом 6, атрибут 4.1.1, бинарное отношение 6.2, встроенный оператор 8, исполнение предиката 5, исполнение предложения 5, конструктор 4.1.3, мир 4.1, предложение 5, простой атом 6.1, '!' 8.

A.5.1.2 Вызовы функций

В составе собственных предложений разрешается использовать вызовы функций. «Вызовом функции» называется синтаксическая конструкция, имитирующая вызов функции, возвращающей некоторое значение — терм.

вызов_функции =
[конструктор_или_атрибут] префикс простой_атом

В ходе трансляции вызов функции вида «$p(A_1,A_2,\ldots,A_n)$» преобразуется в вызов предиката «$p(R,A_1,A_2,\ldots,A_n)$», где R — некоторая уникальная переменная, обозначающая результат функции и помещаемая на место транслируемого вызова функции.

В случае если вызов функции используется в составе некоторой подцели S_a, вызов предиката S_b, соответствующий этому вызову функции, добавляется в состав предложения перед подцелью S_a. При этом гарантируется, что подцель S_b будет помещена перед любой другой подцелью, в состав

которой войдет переменная R, обозначающая результат рассматриваемого вызова функции, но после всех подцелей, находившихся перед подцелью S_a в исходном предложении.

Если вызов функции используется в заголовке предложения вида

Заголовок:– Конъюнкция.

и не является составной частью другого вызова функции, вызов предиката S, соответствующий этому вызову функции, добавляется в состав предложения перед конъюнкцией подцелей «Конъюнкция», перед любыми другими подцелями, построенными в ходе трансляции вызовов функций в этой конъюнкции подцелей:

Заголовок:– S, Конъюнкция.

Если вызов функции используется в заголовке предложения, подцели которого соединены логическими связками «или» —

Заголовок:–
Конъюнкция$_1$
or
Конъюнкция$_2$
...
or
Конъюнкция$_m$.

— и не является составной частью другого вызова функции, вызов предиката S, соответствующий этому вызову функции, добавляется в состав предложения несколько раз — после заголовка предложения и после каждой связки «или»:

Заголовок:–
S,
Конъюнкция$_1$
or
S,
Конъюнкция$_2$
...
or
S,
Конъюнкция$_m$.

При этом гарантируется, что подцель S будет помещаться перед подцелями соответствующей конъюнкции, а также любыми другими подцелями, построенными в ходе трансляции вызовов функций в этой конъюнкции.

Если вызов функции FC_1 в заголовке предложения входит в состав другого вызова функции FC_2, в этом случае FC_1 рассматривается как вызов функции, входящий в состав подцелей предложения, поставленных в соответствие вызову функции FC_2.

Ссылки: вызов предиката 5.1.1, заголовок предложения 5, значение терма 3, или 5.1, конструктор или атрибут 5.1.1, конъюнкция 5.1, переменная 2.1.1, подцель предложения 5.1.1, предложение 5, префикс 5.1.1, простой атом 6.1, собственное предложение 5.1, терм 3, **or** 2.1.2.

A.5.1.3 Примеры собственных предложений

Правильно построенные собственные предложения:

```
clause_1(M,N,J):–
    N * 2 + 1 < 7 – ? f(J),          –– простой ближний вызов
    ('X',key=M) & r(1,N,3),          –– дальний акторный вызов
    console ? write("N=",N).         –– простой дальний вызов


clause_2(K,1,N,L*):–
    check(K, slot ? p(N) ),          –– простой ближний вызов
    @ p(N,7,L).                      –– ближний акторный вызов
```

Ссылки: акторный вызов 5.1.1, ближний вызов 5.1.1, вызов предиката 5.1.1, дальний вызов 5.1.1, простой вызов 5.1.1.

A.5.2 Заголовки внешних предложений

Заголовком внешнего предложения называется объявление внешней подпрограммы, написанной на другом языке программирования. Для определения заголовков внешних предложений служит ключевое слово **external**.

заголовок_внешнего_предложения =
    атом "[" **external** ["!"] "]"

Ограничитель «!» в составе заголовков внешних предложений используется для обозначения детерминированности внешних подпрограмм.

В случае исполнения внешнего предложения осуществляется исполнение соответствующей внешней подпрограммы.

В конкретной реализации языка должны быть определены:

1. Правила построения имен, используемых для обозначения символов и внешних подпрограмм вне текста программы.

2. Внутреннее представление и механизм передачи параметров внешних подпрограмм.

3. Механизм вызова внешних подпрограмм.

Пример. Правильно построенные заголовки внешних предложений:

write(T∗) [**external**!]
writef(F,T∗) [**external**!]
repeat [**external**]

Ссылки: атом 6, исполнение предложения 5, ключевое слово 2.1.2, ограничитель 2.1.5, программа 4, символ 2.1.2, **external** 2.1.2.

A.5.3 Объявления внешних вызовов

Объявление внешних вызовов — это предложение, определяющее имя и арность предикатного символа заголовков предложений экземпляра класса, которые могут быть вызваны из другого языка программирования.

Объявление внешних вызовов включает предикатный символ и, если это необходимо, список аргументов в круглых скобках. Никаких дополнительных ограничений на вид аргументов в объявлении внешних вызовов не накладывается, однако в конкретных реализациях языка использование аргументов различных видов может служить для передачи транслятору некоторой вспомогательной информации.

Завершается объявление внешнего вызова ограничителем «?».

объявление_внешних_вызовов =
    символ [“(” [выражения] “)”] “?”

Объявления внешних вызовов используются только на этапе трансляции программы и в дальнейшем не влияют на ее исполнение.

Пример. Правильно построенные объявления внешних вызовов:

request(Code) ?	event() ?
goal ?	call(_,A,7*[1,B,_],C) ?

Ссылки: выражения 3.3, заголовок предложения 5, исполнение программы 4, мир 4.1, ограничитель 2.1.5, предложение 5, программа 4, символ 2.1.2.

A.6 Атомарные формулы

Атомарными формулами (атомами) в языке являются следующие обозначения:

атом =
    простой_атом |
    бинарное_отношение |
    объявление_функции

Ссылки: бинарное отношение 6.2, объявление функции 6.3, простой атом 6.1.

A.6.1 Простые атомы

Простой атом — это функтор с соответствующим количеством аргументов или недоопределенное множество:

простой_атом =
    символ ["(" [выражения ["*"]] ")"] |
    недоопределенное_множество

Последний аргумент атомарной формулы может быть помечен «*», если он является переменной. В этом случае атомарная формула обозначает предикат с переменным числом аргументов («предикат переменной арности»), а помеченная переменная — список аргументов, не определенных явно в составе атомарной формулы. Во время трансляции арность такого предиката неопределена, однако в ходе исполнения программы эта атомарная формула может быть унифицирована, в общем случае, с атомом любой арности большей или равной R−1, где R — арность используемого предикатного символа.

Переменная в атомарной формуле подцели предложения может быть помечена «*» лишь в том случае, если она таким же образом помечена в заголовке предложения и не является анонимной переменной «_».

Рекурсивное (или взаимно-рекурсивное) определение предикатов переменной арности, увеличивающее число передаваемых аргументов, считается синтаксической ошибкой.

Атомарная формула вида

$$A_0\{x_1{:}A_1,x_2{:}A_2,\ldots,x_n{:}A_n|Rest\}$$

эквивалентна

$$''(\{'':A_0,x_1{:}A_1,x_2{:}A_2,\ldots,x_n{:}A_n|Rest\}),$$

где '' — символ, состоящий из пустой цепочки графем.

Пример. Предложение, имитирующее правило 2-го порядка.

Для обозначения данных в примере используются недоопределенные множества, в состав которых входит признак четности «is_even».

```
P{is_even:unknown|Rest}:–
    P{is_even:yes|Rest},
    P{is_even:no|Rest}.
```

Приведенное утверждение означает, что любой предикат P является истинным при любых значениях аргумента, если его истинность удается доказать отдельно для любых четных и любых нечетных значений этого аргумента.

Ссылки: анонимная переменная 2.1.1, атом 6, выражения 3.3, графема 1, данные 3, заголовок предложения 5, исполнение программы 4, недоопределенное множество 3.2.3, переменная 2.1.1, подцель предложения 5.1.1, предложение 5, символ 2.1.2, список 3.2.2, '' 2.1.2.

A.6.2 Бинарные отношения

«Бинарным отношением» называется атомарная формула, состоящая из двух выражений, соединенных оператором отношения:

```
бинарное_отношение =
    выражение оператор_отношения выражение
```

В качестве знаков операций в бинарных отношениях используются ключевое слово **is**, имена встроенных предикатов '==' и ':=', а также некоторые знаки операций сравнения:

оператор_отношения =
is | "==" | ":=" | "<" | ">" | "<>" | "<=" | ">="

Бинарное отношение, в состав которого входит такой знак операции, эквивалентно обозначению вида

символ(аргумент$_1$,аргумент$_2$),

где символ — знак операции, заключенный в апострофы, аргумент$_1$ и аргумент$_2$ — выражения слева и справа от знака операции.

Пример. Правильно построенные бинарные отношения:

X **is** K $*$ sin(N $+$ M)	X $+$ Y $<$ 54 $*$ Z
[1,3,5|_] $==$ [H|R]	A $<>$ B $+$ C
slot_o1 $:=$ {red,green,blue}	75 $>=$? get_item(2500)

Ссылки: атом 6, встроенный предикат 8, выражение 3.3, ключевое слово 2.1.2, символ 2.1.2, **is** 2.1.2.

A.6.3 Объявления функций

«Объявлением функции» называется синтаксическая конструкция, имитирующая объявление функции, возвращающей некоторое значение — терм. Объявления функций разрешается использовать только в качестве заголовков собственных предложений, а также в заголовках внешних предложений.

объявление_функции = простой_атом "=" выражение

В ходе трансляции объявления функций преобразуются в предикаты. В случае если объявление функции используется в качестве заголовка собственного предложения, являющегося фактом, или в составе заголовка внешнего предложения, синтаксическая конструкция

p(A$_1$,A$_2$,...,A$_n$) = E

преобразуется в предикат

p(E,A$_1$,A$_2$,...,A$_n$).

Если объявление функции используется в собственном предложении, содержащем подцели —

$p(A_1,A_2,\ldots,A_n) = E$ **if** Конъюнкция.

– такое предложение преобразуется к виду

$p(R,A_1,A_2,\ldots,A_n)$ **if** Конъюнкция, $R == E$. ,

где R — некоторая уникальная переменная, обозначающая результат функции. Если объявление функции используется в собственном предложении, подцели которого соединены логическими связками «или» –

$p(A_1,A_2,\ldots,A_n) = E$
if
Конъюнкция_1
or
Конъюнкция_2
or
...
Конъюнкция_m. ,

то подцель «$R == E$» добавляется несколько раз — в конце предложения, а также перед каждой связкой «или»:

$p(R,A_1,A_2,\ldots,A_n)$:–
Конъюнкция_1,
$R == E$
or
Конъюнкция_2,
$R == E$
or
...
Конъюнкция_m,
$R == E$.

Пример. Определение функции append.

Определение функции append, добавляющей элементы в конец списка,

append([ ],L) = L.
append([H|L1],L2) = [H | ? append(L1,L2)].

эквивалентно процедуре вида

```
append(L,[ ],L).
append(R0,[H|L1],L2):–
    append(R1,L1,L2),
    R0 == [H | R1].
```

Ссылки: выражение 3.3, заголовок внешнего предложения 5.2, заголовок предложения 5, значение терма 3, переменная 2.1.1, подцель предложения 5.1.1, предложение 5, простой атом 6.1, процедура 5, собственное предложение 5.1, список 3.2.2, терм 3, факт 5.1, **if** 2.1.2, **or** 2.1.2, '==' 3.4.

A.7 Акторный механизм

Акторным механизмом называется стратегия управления, обеспечивающая логическую корректность нейтрализации и повторного доказательства акторов.

Акторный механизм языка обеспечивает возможность актуализации значений общих переменных, копирования производных значений общих переменных, а также обработку исключительных ситуаций.

Ссылки: актор 7.1, актуализация 7.5, копирование производных значений 7.6, нейтрализация актора 7.1, обработка исключительных ситуаций 7.7, общие переменные 7.2, повторное доказательство 7.1.

A.7.1 Акторы

Актором называется подцель доказательства, соответствующая акторному вызову предиката. Построение акторов происходит динамически, во время исполнения программы.

Актор Q называется «вложенным» по отношению к актору P, если доказательство актора Q, результаты которого в данный момент не отменены, происходит (произошло) в ходе доказательства актора P.

Нейтрализацией актора называется отмена всех результатов его доказательства, за исключением созданных в ходе его доказательства миров и результатов доказательства вложенных по отношению к нему акторов — нейтрализация актора не влияет на состояние других (в том числе и вложенных по отношению к нему) акторов программы.

Повторным доказательством актора называется повторение доказательства актора с самого начала. В ходе повторного доказательства актора,

содержащего акторные вызовы предикатов, происходит построение новых (вложенных по отношению к нему) акторов.

Актор, доказательство которого началось, но еще не окончено, называется «активным». Если доказательство актора завершилось успехом и не отменено, актор называется «доказанным». «Нейтральным» называется такой актор, предыдущее доказательство которого отменено, а повторное доказательство еще не началось.

«Вре́менными» акторами называется специальная разновидность акторов, которые не доказываются повторно в случае их нейтрализации. Явное определение вре́менных акторов осуществляется с помощью префикса «&».

Сообществами акторов называются некоторые выделенные группы акторов, возникающие во время исполнения программы. Сообщество акторов C_q называется «вложенным» по отношению к другому сообществу C_p, если в составе C_q существует некоторый активный актор Q, вложенный по отношению к некоторому активному актору P сообщества C_p.

В максимальной и быстрой версиях языка используются «активные» сообщества акторов, обладающие следующим свойством: доказанные акторы, входящие в состав некоторого активного сообщества C_r, считаются активными (независимо от их реальных состояний) при исполнении глобальных операций, вызванных акторами, не входящими в состав сообщества C_r, а также в состав активных сообществ, вложенных по отношению к сообществу C_r.

Примечание. Использование в программе вре́менных акторов может нарушить ее полноту относительно декларативной семантики.

Пример. Определение ограничения целостности базы данных.

Определяемое ограничение целостности обеспечивает наличие пути между любыми двумя вершинами некоторого ориентированного графа, дуги которого хранятся в базе данных в виде пар {sender:X,receiver:Y}. Оно реализовано в виде актора, повторно доказываемого при любой попытке изменить содержимое базы данных.

```
goal:–
    @ database_constraint.          –– В случае повторного
database_constraint:–               –– доказательства актор
    find({sender:A|_}),             –– подтверждает свою
    find({receiver:B|_}),           –– зависимость от содержимого
    path_not_exists(A,B),!,         –– базы данных, чтобы быть
    fail.                           –– вновь повторно доказанным
database_constraint:–               –– при следующем его
    find(_).                        –– изменении — для этого он с
path_not_exists(A,B):–              –– помощью предиката find(_)
    path_exists(A,B,[A]),!,         –– обращается к базе данных
    fail.                           –– и находит в ней некоторую
path_not_exists(_,_).               –– произвольную запись.
```

Рекурсивное условие существования пути между вершинами графа:

```
path_exists(A,A,_):–!.
path_exists(A,B,List):–
    find({sender:A,receiver:C|_}),
    is_not_element(C,List),
    path_exists(C,B,[C|List]).
```

Ссылки: акторный вызов 5.1.1, глобальные операции 7.2, доказательство акторов 7.3, исполнение программы 4, мир 4.1, подцель доказательства 5, префикс 5.1.1, программа 4, fail 8, goal 4.1.4, '!' 8.

A.7.2 Общие переменные

Будем говорить, что актор P «использует» переменную V (или что переменная V «соответствует», «принадлежит» актору P), если переменная V входит в состав подцели доказательства P или какой-либо другой подцели доказательства Q, построенной в ходе:

- текущего доказательства подцели P, если исполнение актора еще не закончено,
- последнего завершившегося успехом доказательства актора P, если исполнение актора закончено, и он не является нейтральным,

не считая тех подцелей доказательства Q, которые были построены в ходе доказательства акторов, вложенных по отношению к P.

«Общей» называется переменная, которая используется (или может быть использована) несколькими акторами.

Каждый актор использует свои собственные («локальные») значения общих переменных, не зависящие от других акторов — в результате замены переменных во время унификации термов происходит замена только тех вхождений переменных, которые соответствуют некоторому конкретному актору.

Производными значениями общих переменных называются значения, которые можно получить из локальных значений некоторого конкретного актора, унифицировав их с соответствующими локальными значениями, принадлежащими:

1. активным акторам, по отношению к которым рассматриваемый актор является вложенным;

2. всем доказанным акторам.

Считается, что набор акторов находится в «согласованном» состоянии, если все эти акторы являются доказанными, и для каждого из них существуют производные значения общих переменных.

Если все акторы программы находятся в согласованном состоянии, существуют «производные значения общих переменных программы» — множество производных значений общих переменных всех акторов программы.

Глобальными операциями (с общими переменными) называются операции, в которых используются локальные значения, принадлежащие нескольким акторам — сопоставление локальных значений общих переменных, актуализация значений общих переменных, копирование производных значений общих переменных.

Ссылки: активный актор 7.1, актор 7.1, актуализация 7.5, вложенные акторы 7.1, доказанный актор 7.1, доказательство акторов 7.3, значение переменной 3.1, исполнение предиката 5, копирование производных значений 7.6, нейтральный актор 7.1, переменная 2.1.1, подцель доказательства 5, программа 4, сопоставление локальных значений 7.3.1, унификация термов 3.4.

A.7.2.1 Построение общих переменных

Общие переменные создаются автоматически во время исполнения программы в составе:

1. значений слотов во время формирования экземпляра класса;

2. подцелей предложений — в аргументах акторного вызова во время первого доказательства соответствующего актора, а также в аргументах конструктора в дальнем вызове предиката перед доказательством этого конструктора.

При построении общих переменных в составе подцелей предложений, выполняются следующие правила:

1. Если в составе аргумента акторного вызова или инициализатора аргумента конструктора (явным образом) задан атрибут, то в качестве соответствующего ему значения слота воспроизводится начальное значение этого слота (созданное во время формирования слота), вместе с соответствующими ему общими переменными (даже если значение этого слота за время, прошедшее после его формирования, уже было конкретизировано).

2. Если в составе аргумента акторного вызова или инициализатора аргумента конструктора явным образом задана некоторая переменная X (созданная в ходе исполнения некоторого предиката на этапе построения копии и переименования переменных соответствующего предложения), и эта переменная является несвязанной, то в качестве значения переменной X используется некоторая (сцепленная с ней) общая переменная — всем таким вхождениям переменной X во всех подцелях исполняемого предложения ставится в соответствие одна и та же (уникальная) общая переменная.

3. В качестве значений (всех остальных) несвязанных переменных Y, обнаруженных в составе аргументов акторного вызова или инициализатора аргумента конструктора, и не соответствующих условиям пунктов 1, 2, используются некоторые уникальные (сцепленные с ними) общие переменные, при этом всем вхождениям таких переменных Y во всех подцелях предложения и в каждой отдельной подцели ставятся в соответствие различные общие переменные.

Ссылки: актор 7.1, акторный вызов 5.1.1, атрибут 4.1.1, дальний вызов 5.1.1, доказательство акторов 7.3, значение переменной 3.1, значение слота 4.1, инициализатор 4.1.2, исполнение предиката 5, исполнение программы 4, конструктор 4.1.3, начальное значение слота 4.1.4, несвязанная

переменная 3.1, общие переменные 7.2, переменная 2.1.1, подцель предложения 5.1.1, предложение 5, слот 4.1, формирование миров 4.1.4, формирование слота 4.1.4.

A.7.2.2 Пример определения общих переменных

В ходе исполнения предложения

```
goal:–
    X == f(1,K,_),
    @ p1(7,X,X,Y,g(K,Z)),
    & p2(X,Y,t(Z,_),K,_).
```

будут созданы общие переменные _1, _2, _3, _4, Y′, K′, Z′, _8, _9, _10, _11, которые будут переданы в качестве аргументов акторных вызовов:

```
@ p1(7,f(1,_1,_2),f(1,_3,_4),Y′,g(K′,Z′)),
где _1 == K′, _3 == K′, _2 == _4;
& p2(f(1,_8,_9),Y′,t(Z′,_10),K′,_11),
где _8 == K′, _9 == _2.
```

Ссылки: акторный вызов 5.1.1, исполнение предложения 5, общие переменные 7.2, goal 4.1.4, '==' 3.4.

A.7.3 Согласование акторов

Исполнение акторных вызовов предикатов осуществляется в соответствии со стандартной стратегией управления, при этом, однако, необходимым условием успешного завершения доказательства любого актора P является существование соответствующих ему производных значений общих переменных, а также производных значений общих переменных всех доказанных акторов программы.

Для того чтобы обеспечить существование искомых производных значений, в момент завершения доказательства актора P осуществляется согласование акторов программы.

«Согласование акторов» включает:

1. сопоставление локальных значений общих переменных акторов;
2. повторное доказательство акторов, нейтрализованных на первом этапе.

Ссылки: актор 7.1, акторный вызов 5.1.1, доказанный актор 7.1, исполнение предиката 5, нейтрализация актора 7.1, общие переменные 7.2, повторное доказательство 7.1, программа 4, производное значение 7.2, сопоставление локальных значений 7.3.1, стандартная стратегия 5.

A.7.3.1 Сопоставление локальных значений

На первом этапе согласования акторов сопоставляются локальные значения общих переменных, принадлежащие различным акторам. В ходе сопоставления локальных значений происходит нейтрализация акторов программы согласно следующим правилам:

1. Если существование искомых производных значений можно обеспечить, нейтрализовав некоторое множество акторов NA, в ходе сопоставления эти акторы должны быть выявлены и нейтрализованы.

2. Если таких множеств несколько, произвольным образом выбирается одно из них.

3. В число нейтрализуемых акторов NA не разрешается включать активные и нейтральные акторы.

4. Доказанный актор Q не может быть включен в число нейтрализуемых акторов NA, если существует не содержащее актор Q подмножество NA, нейтрализация которого обеспечивает существование искомых производных значений.

5. Порядок нейтрализации акторов в языке не определен.

Если обеспечить существование производных значений невозможно, происходит откат доказательства актора P.

В случае успешного завершения сопоставления локальных значений общих переменных, автоматически вызывается повторное доказательство всех нейтральных акторов (за исключением вре́менных).

Согласование акторов считается успешным в том и только в том случае, если завершаются успехом сопоставление локальных значений общих переменных, а также все повторные доказательства нейтральных акторов.

Ссылки: активный актор 7.1, актор 7.1, временный актор 7.1, доказанный актор 7.1, доказательство акторов 7.3, локальное значение 7.2, нейтрализация актора 7.1, нейтральный актор 7.1, общие переменные 7.2, откат программы 7.3.3, повторное доказательство 7.1, программа 4, производное значение 7.2, согласование акторов 7.3.

A.7.3.2 Исполнение повторных доказательств

Порядок исполнения повторных доказательств нейтральных акторов в языке не определен.

В максимальной и быстрой версиях языка допускается параллельное исполнение повторных доказательств акторов NA. При этом соблюдаются следующие правила:

1. Каждый актор R множества NA, а также все акторы, созданные или нейтрализованные в результате рассматриваемого повторного доказательства актора R, образуют сообщество акторов, активное с момента вызова названного повторного доказательства до его завершения.

2. Если в ходе доказательства некоторого актора R множества NA будет нейтрализован актор Q, принадлежащий активному сообществу C_q, по отношению к которому сообщество C_r, соответствующее актору R, является вложенным, актор Q будет включен в состав сообщества C_r.

3. Если после успешного завершения повторного доказательства некоторого актора R множества NA он или один из акторов, созданных или доказанных повторно в результате его доказательства, будет нейтрализован в результате доказательства некоторого другого актора Q множества NA, повторное доказательство которого еще не окончено, сообщества акторов, соответствующие акторам R и Q, объединяются, при этом объединенное сообщество считается активным до завершения повторного доказательства актора Q, а в случае отката доказательства актора Q, подцели R и Q рассматриваются как последовательность подцелей (R,Q).

4. В случае прекращения доказательства актора (в результате отката или возникновения исключительной ситуации), доказательство всех вложенных по отношению к нему акторов прекращается.

Все перечисленные преобразования сообществ акторов являются необратимыми и не отменяются в случае отката повторного доказательства акторов NA.

Доказательство актора P считается успешным в том и только в том случае, если завершаются успехом исполнение соответствующего предиката, а также последующее согласование акторов, в противном случае происходит откат программы.

Примечание. В результате повторного доказательства недетерминированного актора, могут возникать новые точки выбора.

Ссылки: активное сообщество 7.1, актор 7.1, вложенные акторы 7.1, вложенные сообщества 7.1, доказательство акторов 7.3, исключительная ситуация 7.7, исполнение предиката 5, нейтрализация актора 7.1, нейтральный актор 7.1, откат программы 7.3.3, повторное доказательство 7.1, подцель доказательства 5, согласование акторов 7.3, сообщество акторов 7.1.

A.7.3.3 Откат программы

В результате отката логической программы, осуществляется восстановление состояний акторов на момент прохождения последней (неустраненной оператором отсечения) точки выбора (в том числе отмена всех связываний переменных, произошедших в акторах с момента прохождения этой точки, а также отмена нейтрализации и повторных доказательств акторов), однако такое восстановление для всех акторов гарантируется только в минимальной версии языка.

В быстрой и максимальной версиях восстановление гарантируется только для актора, доказательство которого содержит упомянутую точку выбора, и акторов, по отношению к которым он являлся вложенным в момент построения этой точки.

Актор, восстановление состояния которого (осуществляемое вследствие отката программы) было отложено на некоторое время, называется «задержанным».

Задержанные акторы, нейтрализованные с момента прохождения последней неустраненной точки выбора, могут восстанавливаться независимо друг от друга, через некоторые произвольные промежутки времени после окончания отката (если только их состояние не будет восстановлено принудительно, вследствие очередного отката программы). Пока не завершится их восстановление, задержанные акторы могут находиться в любых состояниях «доказан», пройденных ими с момента построения рассматриваемой точки выбора, за исключением состояний, которым соответствуют доказательства, в ходе которых были созданы новые акторы.

Возобновление задержанного восстановления состояний акторов разрешается лишь в том случае, если в программе существует хотя бы один активный актор, еще не прошедший согласование с другими акторами.

Непосредственно в ходе отката прекращают существование все экземпляры классов и акторы, созданные после прохождения последней неустраненной точки выбора.

Ссылки: активный актор 7.1, актор 7.1, вложенные акторы 7.1, доказанный актор 7.1, доказательство акторов 7.3, мир 4.1, нейтрализация актора 7.1, повторное доказательство 7.1, программа 4, связывание переменной 3.1, '!' 8.

A.7.3.4 Пример определения сети акторов

Рассмотрим поведение фрагмента программы

```
goal:–
      @ actor_2(X,Y,Z), @ actor_3(Y),
      @ actor_4(Y), @ actor_5(X,Z), @ actor_1(X).


actor_1(9).
actor_2(F,F,1).     actor_2(9,8,7).
actor_3(2).         actor_3(8).
actor_4(2).         actor_4(8).
actor_5(2,1).       actor_5(9,7).
```

в момент завершения доказательства актора, соответствующего вызову предиката actor_1.

В это время уже будет существовать сеть акторов

<Actors,Common_Variables>,

где Actors = $\{R_1,R_2,R_3,R_4,R_5,R_0\}$ — акторы, соответствующие вызовам предикатов actor_1, actor_2, actor_3, actor_4, actor_5, и корневой актор, Common_Variables = $\{X',Y',Z'\}$ — общие переменные.

В результате сопоставления локальных значений общих переменных, будут нейтрализованы акторы NA = $\{R_2,R_5\}$, хотя возможен и другой вариант: $\{R_3,R_4,R_5\}$. В ходе повторного доказательства акторов NA акторы R_3, R_4 также будут нейтрализованы и доказаны повторно. После завершения доказательства всех нейтральных акторов, доказательство актора R_1 будет объявлено успешным, производные значения общих переменных программы станут $X'=9$, $Y'=8$, $Z'=7$.

Ссылки: актор 7.1, вызов предиката 5.1.1, доказательство акторов 7.3, корневой актор 4.2, нейтрализация актора 7.1, нейтральный актор 7.1, общие переменные 7.2, повторное доказательство 7.1, программа 4, производные значения программы 7.2, сопоставление локальных значений 7.3.1, goal 4.1.4.

A.7.4 Корректное разрушающее присваивание

Разрушающим присваиванием называется изменение производных значений общих переменных, сопровождаемое нейтрализацией и повторным доказательством некоторых зависящих от них акторов.

Для изменения производных значений общих переменных программы непосредственно в ходе доказательства актора P, используется (недетерминированный) встроенный предикат разрушающего присваивания

L := R.

Исполнение этого предиката осуществляется следующим образом:

1. Унифицируются локальные значения аргументов предиката.
2. После выполнения первого шага происходит согласование акторов программы с локальными значениями актора P в соответствии с правилами, определенными в разделе 7.3.

Доказательство предиката считается успешным в том и только в том случае, если завершаются успехом унификация локальных значений его аргументов, а также последующее согласование акторов программы.

Встроенный предикат разрушающего присваивания разрешается использовать с произвольным количеством аргументов:

':='(V_1,...,V_k).

Примечание. Встроенный предикат разрушающего присваивания является недетерминированным потому, что вызываемое им повторное доказательство акторов, в общем случае, может приводить к построению новых точек выбора.

Пример. Использование предиката разрушающего присваивания.

Рассмотрим доказательство предиката goal:

```
goal:–
      @ subgoal_a(X),
      subgoal_b,
      readint(Y),
      X := Y.
subgoal_a(1).
subgoal_a(3).
subgoal_a(5).
```

В ходе доказательства предиката сначала будет использован факт subgoal_a(1), переменная X при этом получит значение 1. Затем будут исполнены подцели subgoal_b и readint(Y) — пользователь введет значение переменной Y, например, Y=5. Когда очередь дойдет до отношения X:=Y, произойдет повторное исполнение актора, соответствующего подцели subgoal_a — на этот раз будет выбран факт subgoal_a(5). Подцели subgoal_b и readint при этом передоказывать не придется.

Ссылки: актор 7.1, встроенный предикат 8, доказательство акторов 7.3, значение переменной 3.1, исполнение предиката 5, локальное значение 7.2, нейтрализация актора 7.1, общие переменные 7.2, переменная 2.1.1, повторное доказательство 7.1, подцель предложения 5.1.1, программа 4, производное значение 7.2, согласование акторов 7.3, унификация термов 3.4, факт 5.1, goal 4.1.4.

A.7.5 Актуализация значений общих переменных

Актуальными значениями общих переменных некоторого актора P называются значения, которые можно получить из локальных значений этого актора, унифицировав их с соответствующими локальными значениями, принадлежащими активным акторам, по отношению к которым актор P является вложенным.

Актуализацией значений общих переменных называется унификация локальных значений общих переменных рассматриваемого актора с соответствующими ему актуальными значениями. Эта операция осуществляется с помощью встроенного предиката

in(V_1,...,V_k).

Предикат in проверяет возможность актуализации значений общих переменных актора. Если актуализация значений возможна, результатом его исполнения становится унификация локальных значений переменных в составе аргументов V_1,...,V_k с соответствующими актуальными значениями, в противном случае исполнение предиката заканчивается неудачей.

Примечание. Актор не может отменять результаты доказательства других активных акторов, поэтому во многих случаях оказывается удобным начинать доказательство актора актуализацией его локальных значений, чтобы использовать актуальные значения во время доказательства.

Пример. Использование предиката in.

Рассмотрим доказательство предиката goal класса

```
class 'TWO_ACTORS' using 'WINDOW' is
shared_data_item
[
goal:–
    @ subgoal_a.
subgoal_a:–
    shared_data_item == 100,
    @ subgoal_b.


subgoal_b:–
    in(shared_data_item),
    write("Shared Data = ",shared_data_item).
]
```

С помощью предиката in осуществляется актуализация локального значения общей переменной shared_data_item, соответствующего актору, построенному в результате исполнения вызова предиката subgoal_b:

```
Shared Data = 100
```

Ссылки: активный актор 7.1, актор 7.1, вложенные акторы 7.1, встроенный предикат 8, вызов предиката 5.1.1, доказательство акторов 7.3, значение переменной 3.1, значение терма 3, исполнение предиката 5, класс 4.1, локальное значение 7.2, общие переменные 7.2, переменная 2.1.1, унификация термов 3.4, **class** 2.1.2, goal 4.1.4, **is** 2.1.2, **using** 2.1.2, '==' 3.4.

A.7.6 Копирование производных значений

Копированием производных значений общих переменных называется унификация локальных значений общих переменных некоторого конкретного актора с соответствующими ему производными значениями.

Встроенный управляющий оператор

```
copy(V1,...,Vk)
```

проверяет возможность построения производных значений общих переменных актора. Если построение производных значений возможно, результатом его исполнения становится унификация локальных значений переменных в

составе аргументов $V_1,\ldots,V_k$ с соответствующими производными значениями, в противном случае исполнение оператора заканчивается неудачей.

Примечание. Использование оператора copy, в отличие от встроенного предиката in, может, в общем случае, нарушить полноту программы относительно ее декларативной семантики.

Пример. Использование оператора copy.

Рассмотрим доказательство предиката goal:

```
goal:–
      @ subgoal_a(X),
      copy(X),
      write("Shared Data = ",X).
subgoal_a(100).
```

В ходе исполнения предиката, чтобы получить полную информацию о производном значении переменной X, с помощью оператора copy копируется ее локальное значение, принадлежащее актору, соответствующему subgoal_a:

```
Shared Data = 100
```

Ссылки: актор 7.1, встроенный оператор 8, встроенный предикат 8, исполнение предиката 5, локальное значение 7.2, общие переменные 7.2, переменная 2.1.1, программа 4, производное значение 7.2, унификация термов 3.4, goal 4.1.4, in 7.5.

A.7.7 Обработка исключительных ситуаций

Исключительной ситуацией называется аварийное состояние вычислительного процесса, зарегистрированное во время исполнения программы. Обработкой исключительной ситуации называются действия, осуществляемые в программе в случае возникновения исключительной ситуации.

Любые исключительные ситуации, возникающие в ходе доказательства, считаются локальными по отношению к самому внутреннему из вложенных акторов, в котором эта ситуация возникла. Доказательство такого актора останавливается и объявляется неудачным, все его результаты отменяются таким образом, как это происходит при откате программы.

После прекращения доказательства актора P в том же самом мире, где был исполнен соответствующий акторный вызов, автоматически вызывается предикат

alarm(Exception),

аргументом которого является «обозначение» обрабатываемой исключительной ситуации — неотрицательное целое число или символ. Если доказательство предиката alarm заканчивается успехом, обработка исключительной ситуации завершается, после чего происходит откат из точки вызова доказательства актора P. Если доказательство заканчивается неудачей, происходит вызов исключительной ситуации с тем же обозначением на следующем уровне вложенности акторов. Если в ходе доказательства предиката alarm возникает новая исключительная ситуация, ее обработка также осуществляется на следующем уровне вложенности акторов. Если в результате возникновения исключительной ситуации прекращается доказательство актора, не являющегося вложенным по отношению к какому-либо актору программы, управление передается встроенному обработчику ошибок, действия которого должны быть определены в конкретной реализации языка.

Исключительная ситуация может быть вызвана из программы с помощью встроенного управляющего оператора

break(Exception),

где Exception — обозначение исключительной ситуации. Если аргумент оператора не является обозначением исключительной ситуации, или если в качестве аргумента использована несвязанная переменная, исполнение оператора завершается неудачей. Допускается использование оператора break без аргумента, в этом случае обозначением исключительной ситуации считается константа 0 (ноль).

Обозначения исключительных ситуаций, вызываемых на системном уровне (за пределами текста программы), должны быть определены в конкретной реализации языка.

Пример. Вызов исключительной ситуации.

```
goal:–
    @ actor.
actor:–
    break(name_of_exception).
alarm(Code):–
    write("Alarm: ",Code).
```

В результате доказательства предиката goal будет напечатано:

Alarm: name_of_exception

Ссылки: актор 7.1, акторный вызов 5.1.1, вложенные акторы 7.1, встроенный оператор 8, встроенный предикат 8, доказательство акторов 7.3, исполнение предиката 5, исполнение программы 4, мир 4.1, несвязанная переменная 3.1, откат программы 7.3.3, переменная 2.1.1, программа 4, символ 2.1.2, числовой литерал 2.1.3, goal 4.1.4.

A.8 Встроенные предикаты и операторы

Встроенными предикатами языка являются goal(), alarm(E) и ''(T), определяемые в тексте программы, а также предопределенные предикаты:

'=='(V_1,...,V_k)	—	унификация термов;
':='(V_1,...,V_k)	—	разрушающее присваивание;
in(V_1,...,V_k)	—	актуализация значений переменных;
fail	—	неудача.

Кроме того, в языке определены встроенные управляющие операторы, использование которых может нарушить полноту программы относительно ее декларативной семантики:

copy(V_1,...,V_k)	—	копирование производных значений;
'!'	—	отсечение;
break(E)()	—	вызов исключительной ситуации;
spypoint(S)	—	обращение к отладчику, результаты исполнения этого оператора должны быть определены в конкретной реализации языка.

Встроенный оператор отсечения устраняет все неисследованные пути (точки выбора), которые встретились с момента начала исполнения предиката, в соответствие которому было поставлено предложение, содержащее оператор.

В программе не допускается определение предикатов, имена которых совпадают с именами предопределенных предикатов и встроенных управляющих операторов. Не разрешается использование таких имен в качестве имен предикатных символов в акторных и дальних вызовах. Синтаксической ошибкой считается также неверное число аргументов в предопределенных предикатах и встроенных управляющих операторах.

Пример. Использование оператора отсечения.

Рассмотрим поведение фрагмента программы

```
goal:–
      write("<1>"),
      subgoal_a,
      write("<7>").
goal:–
      write("<8>").


subgoal_a:–
      write("<2>"),
      subgoal_b, !,                    –– отсечение
      write("<4>"),
      fail.
subgoal_a:–
      write("<6>").


subgoal_b:–
      write("<3>").
subgoal_b:–
      write("<5>").
```

Если убрать оператор отсечения, программа напечатает:

<1><2><3><4><5><4><6><7>

При наличии оператора отсечения будет напечатано:

<1><2><3><4><8>

Ссылки: акторный вызов 5.1.1, актуализация 7.5, дальний вызов 5.1.1, исключительная ситуация 7.7, исполнение предиката 5, копирование производных значений 7.6, переменная 2.1.1, предложение 5, программа 4, разрушающее присваивание 7.4, символ 2.1.2, унификация термов 3.4, alarm 7.7, break 7.7, copy 7.6, goal 4.1.4, in 7.5, '' 2.1.2, ':=' 7.4, '==' 3.4.

Приложение B

Синтаксические правила Акторного Пролога

1.
letter = capital_letter | small_letter
capital_letter =
"A" | "B" | "C" | "D" | "E" | "F" | "G" |
"H" | "I" | "J" | "K" | "L" | "M" | "N" |
"O" | "P" | "Q" | "R" | "S" | "T" | "U" |
"V" | "W" | "X" | "Y" | "Z"
small_letter =
"a" | "b" | "c" | "d" | "e" | "f" | "g" |
"h" | "i" | "j" | "k" | "l" | "m" | "n" |
"o" | "p" | "q" | "r" | "s" | "t" | "u" |
"v" | "w" | "x" | "y" | "z"
digit =
"0" | "1" | "2" | "3" | "4" | "5" | "6" |
"7" | "8" | "9"
letters_and_digits =
[letters_and_digits ["_"]] letter_or_digit
letter_or_digit = letter | digit
2.1.1.
variable =
capital_letter [["_"] letters_and_digits] |
"_" [letters_and_digits]

2.1.2.
symbol =
    small_letter [["_"] letters_and_digits] |
    ' { grapheme } '
2.1.3.
numeric_literal =
    digits ["." digits] [exponent] |
    digits "#" letters_and_digits "#" [exponent] |
    ` grapheme
digits = [digits ["_"]] digit
exponent = e ["+" | "−"] digits
e = "E" | "e"
2.1.4.
segment_of_string = `"' { grapheme | "\" code } `"'
code = "b" | "t" | "n" | "v" | "f" | "r" | numeric_literal
3.
term = simple_term | compound_term | call_of_function
3.1.
simple_term = constant | variable
constant =
    symbol | ["−"] numeric_literal | string_literal
string_literal = [string_literal] segment_of_string
3.2.
compound_term =
    structure | list | underdetermined_set
3.2.1.
structure = symbol "(" expressions ")"
3.2.2.
list = "[" [expressions ["|" tail]] "]"
tail = variable | call_of_function
3.2.3.
elements_of_set =
    [elements_of_set ","] element_of_set
element_of_set = name_of_element [":" expression]
name_of_element = symbol | numeric_literal
key = simple_term | attribute
underdetermined_set =
    [key] "{" [elements_of_set] ["|" tail] "}"

3.3.
expression = [expression additive_operator] addend
addend =
[addend multiplicative_operator] multiplicand
multiplicand = term | attribute | ["−"] "(" expression ")"
expressions = [expressions ","] expression
additive_operator = "+" | "−"
multiplicative_operator = "∗" | "/" | "**div**" | "**mod**"
4.
program = { declaration_of_class | declaration_of_project }
4.1.
declaration_of_class =
"**class**" heading_of_class "**is**" attributes "[" clauses "]"
heading_of_class = name_of_class ["**using**" name_of_class]
name_of_class = symbol
4.1.1.
attributes = { attribute ["=" initializer] }
attribute = symbol
4.1.2.
initializer = term | attribute | constructor
4.1.3.
constructor =
"(" name_of_class { "," attribute ["=" initializer] } ")"
4.2.
declaration_of_project = "**project**" "**is**" constructor
5.
clause =
own_clause |
heading_of_external_clause |
declaration_of_external_calls
clauses = { clause }
5.1.
own_clause = atom [if disjunction] "."
if = ":−" | "**if**"
disjunction = [disjunction or] conjunction
or = "**or**" | ";"
conjunction = [conjunction and] subgoal
and = "," | "**and**"

5.1.1.
subgoal = simple_subgoal | binary_relation | "!"
simple_subgoal =
[[constructor_or_attribute] prefix] simple_atom
constructor_or_attribute = constructor | attribute
prefix = "?" | "@" | "&"
5.1.2.
call_of_function =
[constructor_or_attribute] prefix simple_atom
5.2.
heading_of_external_clause =
atom "[" "**external**" ["!"] "]"
5.3.
declaration_of_external_calls =
symbol ["(" [expressions] ")"] "?"
6.
atom =
simple_atom |
binary_relation |
declaration_of_function
6.1.
simple_atom =
symbol ["(" [expressions ["*"]] ")"] |
underdetermined_set
6.2.
binary_relation =
expression relational_operator expression
relational_operator =
"**is**" | "==" | ":=" | "<" | ">" | "<>" | "<=" | ">="
6.3.
declaration_of_function = simple_atom "=" expression

Приложение C

Термины и определения

Активное сообщество (active community) — см. стр. 169.

Активный актор (active actor) — актор, доказательство которого уже началось, но еще не окончено.

Актор (actor) — подцель доказательства (построенная во время исполнения программы), соответствующая некоторому акторному вызову предиката.

Акторный вызов предиката (actor call of predicate) — тип вызова предиката, обозначаемый с помощью акторных префиксов «@» и «&». Акторные вызовы предикатов служат для построения акторов во время исполнения программы.

Акторный механизм (actor mechanism) — стратегия управления, обеспечивающая логическую корректность нейтрализации и повторного доказательства акторов.

Акторный префикс (actor prefix) — составная часть акторных вызовов предикатов — префикс «@» или «&».

Актуализация значений общих переменных (actualization of values of common variables) — унификация локальных значений общих переменных некоторого конкретного актора с соответствующими актуальными значениями.

Актуальное значение (actual value) — значение общей переменной некоторого актора P, которое можно получить из локального значения этой переменной, унифицировав его с соответствующими локальными значениями, принадлежащими активным акторам, по отношению к которым актор P является вложенным.

Анонимная переменная (anonymous variable) — переменная «_». Считается, что все анонимные переменные в тексте программы являются некоторыми уникальными, однократно использованными именами.

Аргумент (argument) — 1. выражение, заданное в предикате или составном терме; 2. в составе конструктора: пара «атрибут= инициализатор».

Арность (arity) — количество аргументов.

Атомарная формула, атом (atomic formula, atom) — предикат, используемый в предложении в качестве элементарного объекта, обладающего значением истинности.

Атрибут (attribute) — имя слота; объявляется во всех классах, в которых используется соответствующий слот. Атрибут self — предопределенный, он обозначает непосредственно тот мир, в котором это имя используется.

Библиотечный модуль (library unit) — элементарный программный модуль (определение класса или определение проекта), записанный в программную библиотеку в результате трансляции.

Бинарное отношение (binary relation) — атомарная формула, состоящая из двух выражений, соединенных оператором отношения.

Ближний вызов предиката (near call of predicate) — вызов предиката, в котором не указан явно мир, где этот вызов должен быть исполнен. Ближний вызов осуществляется в том же самом мире, в котором исполняется предложение, содержащее данный вызов.

Вложенные акторы (nested actors) — см. стр. 168.

Вложенные сообщества (nested communities) — см. стр. 169.

Временный актор (temporary actor) — специальная разновидность акторов; вре́менные акторы не доказываются повторно в случае их нейтрализации. Явное определение вре́менных акторов осуществляется с помощью префикса «&».

Встроенный оператор (built-in command) — управляющий оператор, являющийся составной частью определения языка. Встроенными операторами являются: copy(V_1,...,V_k), '!', break(E)(), spypoint(S).

Встроенный предикат (built-in predicate) — предикат, являющийся составной частью определения языка. Встроенными предикатами являются goal(), alarm(E), ''(T), а также все предопределенные предикаты.

Вызов предиката (predicate call) — синтаксическая конструкция, определяющая экземпляр класса, в котором этот вызов должен быть исполнен, тип вызова (ближний или дальний, простой или акторный), а также атомарную формулу вызова.

Вызов функции (call of function) — синтаксическая конструкция, имитирующая вызов функции, возвращающей некоторое значение — терм. В ходе трансляции вызовы функций преобразуются в вызовы предикатов.

Выражение (expression) — синтаксическая конструкция, обозначающая элемент данных — представляет собой атрибут или видоизмененный терм.

Глобальные операции (global operations) — операции, в которых используются локальные значения, принадлежащие нескольким акторам — сопоставление локальных значений общих переменных, актуализация значений общих переменных, копирование производных значений общих переменных.

Графический символ, графема (graphic character, grapheme) — элемент набора символов ASCII, имеющий визуальное представление в виде отпечатанного знака или пробела.

Дальний вызов предиката (far call of predicate) — вызов предиката, в котором явным образом (с помощью атрибута или конструктора) указан мир, в котором этот вызов должен быть исполнен.

Детерминированный (deterministic) — такой, в результате исполнения которого не возникают новые точки выбора.

Доказанный актор (proven actor) — актор, доказательство которого завершилось успехом и в данный момент не отменено.

Доказательство — 1. (proving) исполнение чего-либо (например, доказательство актора, конструктора, подцели доказательства, подцели предложения, предиката, целевого утверждения); проверка чего-либо (например, доказательство существования согласованного состояния набора акторов); 2. (proof) результаты успешного доказательства чего-либо.

Доказательство актора — 1. (proving of actor) исполнение (возможно, повторное) акторного вызова предиката; 2. (proof of actor) результаты успешного исполнения актора (в том числе, возможно, точки выбора и значения переменных), включая результаты доказательства вложенных по отношению к нему акторов.

Заголовок внешнего предложения (heading of external clause) — разновидность предложения — объявление внешней подпрограммы, написанной на другом языке программирования.

Заголовок предложения (heading of clause) — атомарная формула в начале предложения.

Задержанный актор (suspended actor) — актор, восстановление состояния которого (осуществляемое вследствие отката программы) было отложено на некоторое время. Задержанные акторы являются основой логи-

ческой интерпретации необратимых процессов.

Знак операции (operator) — ограничитель или ключевое слово, специальным образом используемые в составе выражений и атомарных формул.

Значение инициализатора (value of initializer) — значение терма или мир.

Значение лексемы (value of token) — наименование смысловых единиц, создаваемых лексическим анализатором и соответствующих лексемам, обнаруженным в тексте программы. Последовательность значений лексем передается синтаксическому анализатору в качестве результатов работы лексического анализатора.

Значение переменной (value of variable) — 1. Значением лексемы «переменная» считается соответствующая ей последовательность графем. 2. Значением терма «переменная» является значение лексемы «переменная», до тех пор пока переменная (терм) не будет связана с каким-либо элементом данных — константой или составным термом. Значением связанной переменной считается соответствующий элемент данных.

Значение слота (value of slot) — значение терма или мир.

Значение терма (value of term) — элемент данных или, если терм является несвязанной переменной, значение лексемы «переменная».

Иерархия наследования (inheritance hierarchy) — отношение наследования, задаваемое на множестве классов программы. В языке используется одиночное наследование, согласно которому у класса может быть не более одного непосредственного предка.

Инициализатор (initializer) — синтаксическая конструкция, определяющая начальное значение слота.

Исключительная ситуация (exception) — аварийное состояние вычислительного процесса, зарегистрированное во время исполнения программы.

Исполнение вызова предиката (execution of predicate call) — то же что «исполнение предиката».

Исполнение предиката (execution of predicate) — см. стр. 158.

Исполнение предложения (execution of clause) — см. стр. 158.

Исполнение программы (execution of program) — доказательство существования согласованного состояния набора акторов, возникающих во время этого доказательства. Исполнение программы начинается с доказательства ее проекта, которому соответствует корневой актор.

Использование класса (usage of class) — см. стр. 150.

Использование переменной актором (usage of variable by actor) — Будем говорить, что актор P «использует» переменную V (или что пере-

менная V «соответствует», «принадлежит» актору P), если переменная V входит в состав подцели доказательства P или какой-либо другой подцели доказательства Q, построенной в ходе: а) текущего доказательства подцели P, если исполнение актора еще не закончено, б) последнего завершившегося успехом доказательства актора P, если исполнение актора закончено, и он не является нейтральным — не считая тех подцелей доказательства Q, которые были построены в ходе доказательства акторов, вложенных по отношению к P.

Исходный файл (source file) — то же что «программный модуль».

Класс (class) — набор предложений языка, имеющий уникальное имя и входящий в состав иерархии наследования.

Ключ (key) — значение элемента с именем '', заданное в составе недоопределенного множества.

Ключевое слово (keyword) — символическое обозначение, по которому транслятор распознает некоторые синтаксические конструкции.

Комментарий (comment) — часть текста программы, не влияющая на ее исполнение и служащая для документирования и облегчения чтения программы человеком.

Константа (constant) — простой терм, отличный от переменной.

Конструктор (constructor) — утверждение о существовании экземпляра класса. Аргументы конструктора определяют значения слотов экземпляра класса.

Копирование производных значений (copying of derived values) — унификация локальных значений общих переменных некоторого конкретного актора с соответствующими ему производными значениями.

Корневой актор (root actor) — актор, соответствующий целевому утверждению программы (проекту).

Лексема (token) — наименование смысловых единиц, распознаваемых лексическим анализатором в тексте программы. Лексемами являются: переменные, символы и ключевые слова, целые числовые литералы, вещественные числовые литералы, сегменты строк, ограничители.

Локальное значение общей переменной (local value of common variable) — значение общей переменной, соответствующее некоторому конкретному актору и не зависящее от других акторов.

Максимальная относительная погрешность (maximal relative error) — ошибка представления вещественных чисел; определяет максимальное количество значащих цифр мантиссы вещественного числа, воспринимаемое транслятором.

Метапредложение (metaclause) — предложение, в заголовке которого используется предикат с переменным числом аргументов.

Мир (world) — то же что «экземпляр класса».

Наследование (inheritance) — принцип формализации знаний, в соответствии с которым набор предложений экземпляра некоторого класса C включает предложения класса C, а также предложения всех классов, являющихся предками C в иерархии наследования, заданной на множестве классов программы.

Начальное значение слота (initial value of slot) — значение слота, созданное во время его формирования.

Недетерминированный (non-deterministic) — имеющий несколько возможных путей исполнения, которые могут быть перечислены с помощью отката.

Недоопределенное множество (underdetermined set) — составной терм, построенный из набора (возможно, пустого) элементов, заключенного в фигурные скобки. Элементы недоопределенного множества задаются в виде пар «имя_элемента: выражение». В случае если набор элементов множества не является пустым, в состав множества может быть включен дополнительный компонент — переменная, обозначающая неопределенный остаток (хвост) множества.

Нейтрализация актора (neutralization of actor) — отмена всех результатов доказательства актора, за исключением созданных в ходе его доказательства миров и результатов доказательства вложенных по отношению к нему акторов.

Нейтральный актор (neutral actor) — актор, предыдущее доказательство которого отменено, а повторное доказательство еще не началось.

Необратимый процесс (irreversible process) — процесс перехода физической системы из одного состояния в другое, не допускающий возможность ее возвращения в первоначальное состояние путем прохождения той же последовательности промежуточных состояний в обратном порядке.

Несвязанная переменная (unbound variable) — переменная, не связанная с элементом данных (константой или составным термом).

Обозначение исключительной ситуации (designation of exception) — символ или неотрицательное целое число.

Обработка исключительной ситуации (processing of exception) — действия, осуществляемые в программе в случае возникновения исключительной ситуации.

Общая переменная (common variable) — переменная, которая используется (или может быть использована) несколькими акторами.

Объявление внешних вызовов (declaration of external calls) — предложение, определяющее имя и арность предикатного символа заголовков предложений экземпляра класса, которые могут быть вызваны из другого языка программирования.

Объявление функции (declaration of function) — разновидность атомарной формулы — синтаксическая конструкция, имитирующая объявление функции, возвращающей некоторое значение — терм. В ходе трансляции объявления функций преобразуются в предикаты.

Ограничитель (delimiter) — разновидность лексемы — последовательность из одного или нескольких специальных символов, используемая в синтаксических конструкциях языка.

Оператор отсечения, '!' (cut command) — встроенный управляющий оператор — отсечение устраняет все неисследованные пути (точки выбора), которые встретились с момента начала исполнения предиката, в соответствие которому было поставлено предложение, содержащее оператор.

Откат (backtracking) — возобновление исполнения программы, начиная с последней (неустраненной оператором отсечения) точки выбора. В результате отката происходит восстановление состояний акторов на момент прохождения упомянутой точки выбора (в том числе отмена всех связываний переменных, произошедших в акторах с момента прохождения этой точки, а также отмена нейтрализации и результатов повторных доказательств акторов).

Перекрытие инициализаторов (overriding of initializers) — отмена инициализаторов слотов класса, заданных в классах, являющихся его предками в иерархии наследования.

Переменная (variable) — разновидность лексемы — имя, начинающееся с большой буквы или символа подчеркивания.

Повторное доказательство актора (repeat of actor proving) — повторение доказательства актора с самого начала.

Подцель доказательства (subgoal of proving) — подцель копии собственного предложения, построенной в ходе исполнения некоторого вызова предиката.

Подцель предложения (subgoal of clause) — составная часть собственного предложения — вызов предиката или встроенный оператор, исполнение которого осуществляется в ходе исполнения предложения.

Построение экземпляра класса (generation of class instance) — создание нового экземпляра класса; включает формирование экземпляра класса, а также доказательство предиката goal() во всех сформированных на предыдущем этапе мирах.

Правило (rule) — то же что «собственное предложение».

Предикат (predicate) — синтаксическая конструкция, состоящая из предикатного символа (функтора) и последовательности (возможно, пустой) аргументов. Предикат обозначает некоторую функцию или набор функций (если речь идет о предикате с переменным числом аргументов), принимающих истинностные значения.

Предикатный символ (predicate symbol) — функтор, используемый в качестве имени предиката.

Предикат с переменным числом аргументов (variable arity predicate) — то же что «предикат переменной арности» — предикат, последним аргументом которого является переменная, помеченная ограничителем «$*$». В общем случае, предикат с переменным числом аргументов обозначает набор функций, принимающих истинностные значения.

Предложение (clause) — одна из составных частей определения класса. Различаются собственные предложения, заголовки внешних предложений и объявления внешних вызовов.

Предопределенный предикат (predefined predicate) — предикат, являющийся составной частью определения языка и не определяемый в тексте программы. Предопределенными являются предикаты '$==$'(V_1,...,V_k), '$:=$'(V_1,...,V_k), in(V_1,...,V_k), fail.

Принадлежать актору (belong to actor) — см. «использование переменной актором».

Проверка вхождения (occur check) — проверка, предотвращающая связывание переменной с составными термами, содержащими эту переменную.

Программа (program) — объединение некоторого количества определений классов и определения проекта. Декларативная семантика программы, в которой не используются средства управления, эквивалентна декларативной семантике соответствующей программы без акторных префиксов. Такая программа может быть однозначно представлена в виде формулы логики предикатов первого порядка.

Программная библиотека (program library) — файл, в котором хранятся оттранслированные программные модули.

Программный модуль (program unit) — фрагмент программы, который может быть оттранслирован независимо от остальных.

Проект (project) — целевое утверждение программы — некоторый конструктор экземпляра класса.

Производное значение актора (derived value of actor) — значение некоторой общей переменной актора P, которое можно получить из локаль-

ного значения этой переменной, унифицировав его с соответствующими локальными значениями, принадлежащими некоторым другим акторам программы: активным акторам, по отношению к которым актор P является вложенным, а также всем доказанным акторам.

Производные значения программы (derived values of program) — множество производных значений общих переменных всех акторов программы. Производные значения общих переменных программы существуют, если все акторы программы находятся в согласованном состоянии.

Простой атом (simple atom) — простейшая разновидность атомарной формулы — функтор с соответствующим количеством аргументов или недоопределенное множество.

Простой вызов предиката (simple call of predicate) — тип вызова предиката, обозначаемый префиксом «?» или отсутствием префикса.

Простой терм (simple term) — элементарная синтаксическая конструкция, обозначающая данные. В качестве простых термов используются переменные, символы, целые числа, вещественные числа, строковые литералы.

Процедура (procedure) — набор предложений класса с одинаковыми предикатными символами атомарных формул заголовков.

Разделитель (separator) — один или несколько символов алфавита языка, разделяющих лексемы в тексте программы. Разделителями являются комментарии, а также пробелы и управляющие символы, не входящие в состав лексем и комментариев.

Разрушающее присваивание (destructive assignment) — изменение производных значений общих переменных, сопровождаемое нейтрализацией и повторным доказательством некоторых зависящих от них акторов. Акторный механизм языка гарантирует корректность программы, в которой используется разрушающее присваивание.

Расширенные цифры (extended digits) — цифры и буквы от «A» до «Z» (от «a» до «z»), используемые для записи мантиссы числовых литералов с основанием.

Реагирующие системы (reactive systems) — системы, взаимодействующие с окружающей средой: изменяющие свое внутреннее состояние под воздействием сообщений извне и посылающие ответные сообщения.

Связывание переменной (binding of variable) — замена всех вхождений переменной некоторым элементом данных. В Акторном Прологе область действия операции связывания переменной всегда ограничена множеством (локальных) значений переменных, принадлежащих некоторым конкретным акторам.

Сегмент строки (string segment) — лексема, обозначающая цепочку графических и управляющих символов.

Сеть акторов (net of actors) — набор акторов, связанных общими переменными.

Символ (symbol) — разновидность лексемы — имя, начинающееся с маленькой буквы или заключенное в апострофы.

Слот (slot) — составная часть экземпляра класса, характеризуемая именем и значением. Именем слота является некоторый атрибут, значением слота может быть либо значение терма, либо мир.

Собственное предложение (own clause) — разновидность предложения — логическое правило в составе класса, состоящее из заголовка и последовательности (возможно, пустой) подцелей.

Согласование акторов (coordination of actors) — попытка согласовать локальные значения общих переменных акторов программы; включает сопоставление локальных значений общих переменных акторов, а также повторное доказательство некоторых акторов, нейтрализованных на первом этапе.

Согласованное состояние набора акторов (consistent state of actors) — такое состояние набора акторов, когда все они являются доказанными, и для каждого из них существуют производные значения общих переменных.

Сообщество акторов (community of actors) — множество акторов; сообщество акторов может быть объявлено «активным».

Соответствие переменной актору (correspondence between variable and some actor) — то же, что «использование переменной актором».

Сопоставление локальных значений (comparison of local values) — первый этап согласования акторов; в ходе сопоставления локальных значений допускается нейтрализация некоторых акторов программы.

Составной терм (composite term) — разновидность обозначений элементов данных; составными термами являются структуры, списки и недоопределенные множества.

Специальный символ (special symbol) — графический символ, используемый для построения ограничителей.

Список (list) — составной терм, построенный из последовательности (возможно, пустой) аргументов, заключенной в квадратные скобки. В случае если последовательность аргументов списка не является пустой, в его состав может быть включен дополнительный компонент — переменная, обозначающая остаток (хвост) списка.

Средства управления (control facilities) — синтаксические средства языка, не имеющие декларативной семантики: управляющие операторы, заголовки внешних предложений, акторный префикс «&».

Стратегия управления (control strategy) — алгоритм управления исполнением программы, определяющий порядок выбора предложений программы и порядок исполнения подцелей в предложениях. Для доказательства каждого отдельного актора в языке используется стандартная стратегия управления, соответствующая текстуальному упорядочению процедур и вызовов (используется так называемый «поиск слева направо в глубину с возвратом»), при этом нейтрализация и повторное доказательство акторов может обеспечить любой необходимый порядок доказательства подцелей.

Строковый литерал (string literal) — разновидность простого терма — последовательность сегментов строки, обозначающая цепочку графических и управляющих символов.

Структура (structure) — составной терм, построенный из функтора и последовательности одного или более аргументов, заключенной в круглые скобки.

Сцепление переменных (chaining of variables) — отождествление (несвязанных) переменных; любое связывание одной из сцепленных переменных автоматически вызывает такое же связывание всех сцепленных с ней переменных. В Акторном Прологе область действия операции сцепления переменных всегда ограничена множеством (локальных) значений переменных, принадлежащих некоторым конкретным акторам.

Терм (term) — обозначение элемента данных.

Точка выбора (backtrack point) — неисследованный путь, по которому может пойти исполнение программы в случае отката.

Унификация (unification) — операция сравнения (отождествления) нескольких формул, связывающая переменные в составе формул сопоставленными с ними подформулами. Унификация (несвязанных) переменных друг с другом приводит к сцеплению этих переменных.

Управляющий оператор (control command) — синтаксическая конструкция, выражающая целостное законченное действие, реализуемое во время исполнения программы и способное нарушить ее полноту относительно декларативной семантики.

Управляющий символ (control character) — элемент набора символов ASCII — возврат на одну позицию, горизонтальная табуляция, перевод строки, вертикальная табуляция, перевод формата или возврат каретки.

Факт (fact) — собственное предложение, в составе которого отсутствуют подцели.

Формирование программы (formation of program) — сборка программы из библиотечных модулей, сопровождаемая проверкой их синтаксической правильности.

Формирование слота (formation of slot) — операция создания слота. Одновременно с созданием каждого слота, если для этого слота задан инициализатор, формируется начальное значение слота. Все слоты, не получившие значения в ходе своего создания, получают в качестве начальных значений уникальные общие переменные.

Формирование экземпляра класса (formation of class instance) — первый этап построения экземпляра класса, включающий построение соответствующего пространства поиска, а также формирование слотов экземпляра класса.

Функтор (functor) — имя (символ), которому приписана некоторая арность (число аргументов).

Хвост (tail) — остаток списка или недоопределенного множества.

Целевое утверждение (goal statement) — утверждение, доказательство которого является целью исполнения программы.

Числовой литерал (numeric literal) — лексема, обозначающая числовое значение. Числовые литералы бывают целые и вещественные (плавающие).

Экземпляр класса, мир (class instance, world) — конкретное применение класса; составная часть пространства поиска исполняемой программы. Экземпляр класса характеризуется набором предложений соответствующего класса и его предков, а также набором слотов, доступных во всех этих предложениях. Построение экземпляра класса осуществляется в результате доказательства утверждения о его существовании (конструктора).

Элементарный программный модуль (simple program unit) — определение класса или определение проекта.

Элемент данных (data item) — обозначенная группа данных, обрабатываемая как единое целое.