РОССИЙСКАЯ АКАДЕМИЯ НАУК
ИНСТИТУТ ПРОБЛЕМ ИНФОРМАТИКИ

Методы синтеза канонических моделей, предназначенных для достижения семантической интероперабельности неоднородных источников информации

Л.А. Калиниченко, С.А. Ступников, Н.А. Земцов

Москва
ИПИ РАН
2005

УДК 007:681.3

Калиниченко Л.А., Ступников С.А., Земцов Н.А.
Методы синтеза канонических моделей, предназначенных для достижения семантической интероперабельности неоднородных источников информации. — М.: ИПИ РАН, 2005. — 84 с. — ISBN 5-02-032863-4.

Цель работы состоит в создании формальных методов определения семантических моделей представления неоднородной информации и их уточнений в процессе интеграции неоднородных информационных источников и композиционного проектирования информационных систем.

Методология разработки состоит в создании ядра канонической модели, а также расширений ядра для выражения конкретных моделей данных. При этом корректность построения расширений устанавливается с помощью автоматизированных формальных методов.

Разработаны методы синтеза канонической модели для структурированных моделей данных, объектных моделей и процессных моделей.

ISBN 5-02-032863-4

Содержание

Введение

Настоящий период развития информационных технологий (ИТ) характеризуется взрывоподобным процессом создания разнообразных моделей представления информации. Это развитие происходит как в рамках конкретных распределенных инфраструктур (таких как архитектуры OMG (в частности, архитектуры, движимые моделями представления информации (MDA)), архитектуры семантического Web и Web сервисов, архитектуры электронных библиотек как коллективных хранилищ информации в различных предметных областях, архитектуры информационных грид), так и в стандартах языков и моделей данных (таких как, например, ODMG, SQL, UML, стеки XML и RDF моделей данных), процессных моделей и моделей потоков работ, семантических моделей (включая онтологические модели и модели метаданных), моделей цифровых репозиториев данных и знаний в конкретных областях науки (например, виртуальные обсерватории в астрономии).

Этот процесс сопровождается другой тенденцией – накоплением использующих подобные модели информационных компонентов и сервисов, число которых экспоненциально растет. Этот рост вызывает все увеличивающуюся потребность интеграции модельно неоднородных компонентов и сервисов в различных применениях, а также их повторного использования и композиции для реализации новых информационных систем. Указанные тенденции противоречивы: чем больше разнообразие применяемых моделей в различных компонентах и сервисах, тем более сложными становятся проблемы их интеграции и композиции. Эти тенденции не новы, но с течением времени разнообразие различных моделей и их сложность растет вместе с ростом потребности достижения интеграции и композиции разномодельных компонентов и сервисов. Масштабы этих явлений, определяющих возможности конструирования распределенных информационных систем в различных областях, повторного использования, трейдинга и композиции компонентов, достижения их семантической интероперабельности (совместной работы в конкретных применениях), интеграции неоднородных информационных источников, являются достаточной мотивацией для исследования и разработки адекватных методов оперирования разнообразными моделями представления информации. Основу этих методов составляет понятие канонической информационной модели, служащей в качестве общего языка, «эсперанто», для адекватного выражения семантики разнообразных информационных моделей, окружающих нас.

В лаборатории композиционных методов проектирования инфор-

мационных систем ИПИ РАН проблеме синтеза канонических моделей уделяется значительное внимание на протяжении многих лет. Методы синтеза канонических моделей развивались в лаборатории по мере развития моделей представления информации в ИТ. Можно выделить три периода в этом развитии:

- период структурированных моделей данных (доминировавший до середины 80х годов 20 столетия);
- период объектных моделей и интероперабельного конструирования систем (начался в конце 80х годов);
- период поведенческих, процессных моделей (начался в середине 90х годов).

Этим трем периодам сопоставляются различные методы отображения моделей данных и создания канонических моделей систем интеграции информации, которые разрабатывались в ИПИ РАН.

В первый период были введены основополагающие определения эквивалентности состояний баз данных, схем баз данных и моделей данных для того, чтобы при построении отображений разнообразных структурированных моделей данных в каноническую сохранялись операции и не было потери информации [19, 1]. Каждая модель данных при этом определялась синтаксисом и семантикой двух языков – языка определения данных (ЯОД) и языка манипулирования данными (ЯМД). Основным принципом отображения произвольной исходной модели данных в целевую модель (каноническую) явился *принцип коммутативного отображения моделей данных*, согласно которому сохранение операций и информации исходной модели данных при ее отображении в каноническую достигается при условии, что диаграмма отображения ЯОД (схем) и диаграмма отображения ЯМД (операторов) являются коммутативными [1]. При этом необходимо, чтобы в диаграмме отображения схем отображение пространств состояний баз данных в отображаемых моделях было биективным. Вторым, не менее важным принципом отображения, был *принцип расширения целевой модели данных*. Согласно этому принципу, отображение осуществляется не непосредственно в целевую модель, а в ее расширение, определяемое аксиоматически, так чтобы целевая модель, расширенная набором аксиом, стала эквивалентной исходной модели. Наконец, третьим принципом явился *принцип синтеза канонической модели*, согласно которому фиксируется ее ядро, аксиоматические расширения которого конструируются эквивалентно всевозможным исходным моделям данных, после чего объединения всех

таких расширений вместе с ядром составляют результат синтеза канонической модели.

На основании названных принципов был разработан процесс конструирования отображений моделей данных, в котором в качестве формализма (метамодели) использовалась денотационная семантика, позволявшая демонстрировать коммутативность диаграмм отображения моделей данных [1]. Такой формализм оказался оправданным для точной спецификации связи семантики ЯОД и ЯМД разнообразных моделей данных с их синтаксисом. Доказательство коммутативности диаграмм отображения ЯОД достигалось методом струтурной индукции, а доказательство комутативности диаграмм отображения операторов реализовалось на основе правил эквивалентного преобразования функций метамодели.

Применение метода отображения моделей данных в этот период было продемонстрировано для случая, когда в качестве ядра концептуальной модели использовалась комбинация реляционной и слабоструктурированной модели данных, а в качестве исходных моделей использовались наиболее известные в тот период разнообразные модели данных. В результате этого процесса были построены необходимые отображения и была синтезирована каноническая модель данных. В целом, при относительно небольшом числе исходных моделей данных, используемых в распределенной системе, этот подход, основанный на конструировании отображений моделей данных и доказательстве их коммутативности вручную, был вполне приемлемым и с практической точки зрения.

Во второй период, когда наряду с объектными моделями данных (которые сами стали расширяемыми) и идеями интероперабельности [2], появились новые формальные языки и методы разработки программ (исчисление уточнений, разработка программ на основе их пошаговых уточнений), описанный выше метод отображения моделей данных и построения канонических моделей был видоизменен следующим образом. В качестве формализма (метамодели) метода вместо денотационной семантики была применена Нотация Абстрактных Машин (AMN), позволяющая определять теоретико-модельные спецификации в логике первого порядка и осуществлять доказательство факта уточнения спецификаций [6, 7]. Теория уточнений позволила развить основополагающие определения отношений между типами данных, схемами данных, моделями данных (сформулированные в первый период) так, чтобы вместо эквивалентности соответствующих спецификаций, можно было рассуждать о их уточнении [21]. Говорят, что спецификация A уточняет спецификацию D, если A можно использовать вместо D так, что пользователь D не замечает этой замены.

Наличие специальных инструментов для AMN (B-технология) позволяет осуществлять доказательство коммутативности отображений полу-автоматически: необходимые для доказательства уточнений теоремы генерируются B автоматически, а их доказательство (в общем случае) реализуется с помощью человека. В одной из работ [20] было показано, как характерный для объектной модели данных ODMG тип связи отображается в тип канонической модели данных с проведением всех необходимых доказательств.

В третий период основное внимание уделяется процессным моделям, необходимым для описания деятельности различных организаций при решении соответствующих им задач. Например, модели виртуальных организаций основаны на композиции процессов реальных организаций, вовлеченных в сферу деятельности виртуальной организации. Трейдинг процессов и композиции процессов, реализующих заданный процесс (что является одной из важнейших задач в семантическом Web или в мобильных системах), является другим примером. Процессы реализуются как потоки работ. Большое число разнообразных Систем Управления Потоками Работ (СУПР) разработаны на коммерческой основе. В основе используемых ими процессных языков применяются разнообразные понятия и парадигмы, несовместимые между собой. Спецификация потока работ – сложная конструкция, интегрированная со спецификациями других типов (например, объектных).

Отображение процессов при синтезе их канонической модели требует сохранения семантики одновременного поведения (concurrency). Основная проблема синтеза канонической модели процессов заключается в том, что общей теории одновременного поведения не существует. Как показали ранние исследования [22], процессные алгебры не обладают достаточными выразительными возможностями, чтобы служить ядром канонической модели процессов. Вместе с тем совмещение двух требований – охвата канонической процессной моделью требуемой полноты интерпретации разнообразных моделей потоков работ с доказательностью правильности интерпретации в канонической процессной модели любой модели потоков работ – в течение некоторого времени оставалось трудно достижимым. Лишь недавно была обнаружена возможность интерпретации одновременных событий, характерных для процессных моделей, в логике, в Нотации Абстрактных Машин. Были построены алгоритмы отображения процессных спецификаций в AMN [14, 33]. Такой подход позволяет конструировать доказательные уточнения процессных спецификаций, используя B-технологию. Это достижение является необходимой предпосылкой для коммутативного отображения процессных моделей. Одновремен-

но многообразие моделей потоков работ удалось описать в виде образцов потоков работ [4]. Благодаря этим двум событиям, открылась возможность выбора ядра канонической процессной модели и построения ее расширений, уточняемых различными шаблонами потоков работ. Таким образом, открылся путь к синтезу канонической модели процессов, что и было сделано в рамках настоящей работы.

Текст препринта организован следующим образом. В разделе 1 дан обзор результатов синтеза расширяемой канонической модели и методов коммутативного отображения структурированных моделей данных. В разделе 2 подходы, описанные в разделе 1, расширяются техникой уточнения, которая была применена к объектным моделям данных. Раздел 3 посвящен синтезу расширяемой канонической процессной модели. В разделе 4 рассматривается техника построения уточняющих расширений для процессной модели. В заключении обобщен итог полученных результатов, и обсуждены перспективы применения описанных методов.

Настоящая работа выполнена при поддержке Программы фундаментальных исследований "Фундаментальные основы информационных технологий и систем" Отделения информационных технологий и вычислительных систем РАН (ОИТВС РАН), а также при частичной поддержке Российского фонда фундаментальных исследований, грант 03-01-00821.

1 Отображение и канонизация структурных моделей данных

1.1 Принципы синтеза канонической модели

В данном разделе рассматривается подход к строгому определению моделей данных и использованию их как формальных объектов в процессе построения отображений моделей данных (МД) в контексте интеграции неоднородных баз данных [1, 2].

В каждой системе управления базами данных модель данных полностью определяется семантикой языка определения данных (ЯОД) и языка манипулирования данными (ЯМД). При переходе от некоторой специфической (исходной) МД к канонической (целевой) модели необходимо сохранение информации и операций над ней. Для этого исходная МД должна быть эквивалентным образом отображена в каноническую модель. Понятие *эквивалентности моделей данных* определяется следующим образом. Состояния базы данных в исходной модели и целевой модели эквивалентны, если они отображают-

ся в одно и то же состояние в *абстрактной метамодели*. Предполагается, что эквивалентные состояния базы данных представляют один и тот же набор фактов. Схемы базы данных эквивалентны, если они порождают равномощные множества состояний базы данных, между которыми существует биекция, ставящая в соответствие друг другу эквивалентные состояния базы данных. Две модели данных *эквивалентны* если любой схеме базы данных в одной модели может быть сопоставлена эквивалентная схема в другой модели и наоборот, причем каждая из МД обеспечивает полный набор операторов ЯМД. Основные принципы синтеза канонической модели данных выглядят следующим образом.

Принцип аксиоматического расширения моделей данных. Каноническая модель должна быть расширяемой. Расширение канонической модели происходит при рассмотрении каждой новой модели данных аксиоматически: целевая МД расширяется путем добавления к ее ЯОД набора аксиом, определяющих логические зависимости данных исходной модели в терминах целевой модели. Полученное расширение должно быть эквивалентно исходной модели.

Принцип коммутативного отображения моделей данных. При отображении исходной МД в каноническую необходимо сохранение информации и операторов. Это требование достигается если отображение моделей данных является коммутативным.

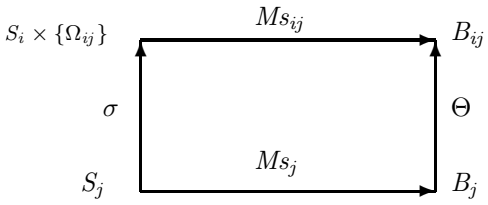
Множество всех схем, выразимых в ЯОД модели данных M_i, обозначается S_i, множество операторов ЯМД модели M_i обозначается O_i. *Пространство допустимых состояний*, выразимых в M_i, обозначается B_i.

$Ms_i : S_i \to B_i$ есть семантическая функция ЯОД M_i.

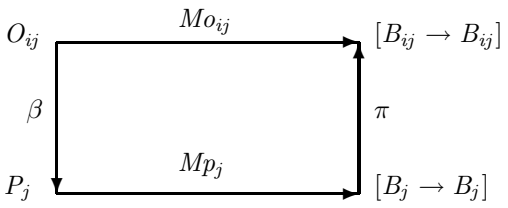
$Mo_i : O_i \to [B_i \to B_i]$ есть семантическая функция ЯМД M_i.

Отображение $f =< \sigma, \theta, \beta >$ модели M_j в расширение M_{ij} модели M_i коммутативно, если выполняются следующие условия:

— диаграмма отображения схем является коммутативной:



— диаграмма отображения операторов ЯМД является коммутативной:



— отображение θ биективно.

Здесь Ω_{ij} обозначает множество схем аксиом, выражающих зависимости данных модели M_j в терминах M_i; P_j обозначает последовательность операторов (процедуру) ЯМД модели M_j.

Принцип синтеза унифицирующей канонической модели данных. Синтез канонической модели данных есть процесс построения расширений ядра канонической модели данных, эквивалентных различным моделям данных, включаемым в среду; а также процесс слияния этих расширений с канонической моделью. В создаваемой согласно этому принципу унифицирующей канонической модели данных разнообразные исходные модели данных имеют однородное эквивалентное представление.

Аксиоматическое расширение целевой модели означает, что семантика операторов ЯМД должна быть соответствующим образом преобразована для того, чтобы сохранялись аксиомы расширения.

1.2 Денотационная семантика как метамодель данных

Основной мотив формального определения моделей данных заключается в том, что таким образом возможно получить компактные и точные описания, позволяющие обращаться с различными моделями данных как с математическими объектами. Метаязык, используемый для формального определения моделей данных, называется метамоделью данных (ММД). Язык ММД должен быть достаточно общим, т.е. независимым от понятий различных моделей данных; должен обладать способностью точного выражения семантических свойств различных моделей данных, их сходства и различия на основе одного и того же языка. Формальное определение модели данных предусматривает строгую дисциплину разработки отображений моделей данных, в соответствии с которой построение отображения моделей и доказательство его корректности должны проводиться одновременно.

Денотационная семантика [35] использовалась в качестве формальной ММД для отображений структурных моделей данных [20]. Ос-

новная идея денотационной семантики состоит в определении класса „типов данных“ — областей определения функций (доменов) — как частично-упорядоченных множеств и набора функций (обычно рекусивных) для создания естественной и точно определенной модели вычислений. Отношение $\sqsubseteq$ ("менее определенный или равный") есть частичный порядок на области определения D такой, что для всех d из D выполняются соотношения $\perp \sqsubseteq d$ и $d \sqsubseteq d$. Тип данных (домен) в ММД есть множество, частично упорядоченное отношением $\sqsubseteq$.

Абстрактная метамодель включает следующие компоненты:

- множество элементарных доменов $D_1, D_2, ...,$ соответствующих первичным множествам объектов;
- операции конструирования сложных доменов (типов данных) на основе более простых доменов и средства формального определения типов данных;
- множество типов данных, определенных на основе элементарных доменов при помощи операций конструирования типов данных;
- множество примитивных функций и предикатов, определенных на типах данных;
- множество функциональных форм, используемых для определения новых функций, и средства формального определения функций;
- множество определений функций;
- множество правил эквивалентного преобразования функций.

Каждый тип данных (домен) T строится рекурсивно из типов данных (доменов) $D_i (1 \leq i \leq n)$ при помощи операций конструирования доменов $OP = + \mid \times \mid \rightarrow$ (сумма, произведение и функциональный домен, соответственно) по следующим правилам:
$T = D_i$ (любой домен есть тип данных), $T = T^\star$ (список), $T = T^n$, $T = T\ OP\ T$. В выражениях конструирования доменов могут использоваться скобки: $T = (T\ OP\ T)$. Для удобочитаемости функциональный домен помещается в квадратные скобки: $T = [T \rightarrow T]$.

Функциональная форма (условная форма, функциональная подстановка, композиция) в ММД есть выражение, обозначающее функцию. Эквивалентные преобразования функций используются для доказательства коммутативности диаграмм отображения операторов при отображении МД. Ниже приводятся примеры основных преобразований [9].

Здесь *f,g,h* обозначают призвольные функции, *p,q* – обозначают произвольные предикаты.

$$\bar{\perp} \circ f \equiv f \circ \bar{\perp} \equiv \bar{\perp} \tag{1}$$

$$(p \longrightarrow f, g) \circ h \equiv p \circ h \longrightarrow f \circ h, g \circ h \tag{2}$$

$$h \circ (p \longrightarrow f, g) \equiv p \longrightarrow h \circ f, h \circ g \tag{3}$$

$$p \longrightarrow (p \longrightarrow f, g), h \equiv p \longrightarrow f, h \tag{4}$$

Полное описание ММД содержится в [1, 20].

1.3 Структура формального определения модели данных

Определение ЯОД модели данных M_i состоит из следующих компонентов:

1. *Абстрактного синтаксиса ЯОД* как домена всех схем, выразимых в модели данных;
2. *Семантических доменов ЯОД* как типов данных, представимых в модели данных;
3. *Схем семантических функций ЯОД*;
4. *Функций интерпретации конструкций ЯОД в семантических доменах.*

Определение семантики ЯМД M_i состоит из следующих компонентов:

1. *Абстрактного синтаксиса ЯМД* как домена всех операторов, выразимых в M_i;
2. *Семантических доменов ЯМД*, включающих семантические домены ЯОД и типы данных, характеризующие состояние программы на ЯМД;
3. *Схем семантических функций ЯМД*;
4. *Функций интерпретации операторов ЯМД*, выражающих изменения базы данных, вызванные исполнением операторов ЯМД.

1.4 Процесс построения коммутативного отображения модели данных

Общая схема процесса построения коммутативного отображения структурной модели данных изображена на рисунке 1.1. Процесс решает две задачи: 1) построения расширений ядра канонической модели, эквивалентных исходным моделям данных; 2) разработки и верификации интерпретации канонического ЯМД исходным ЯМД. Данный подход интенсивно использовался в начале 80х годов. Применения данного процесса и соответствующие примеры описаны в [1].

1.5 Каноническая модель для структурных исходных моделей данных

На основании систематического использования метода коммутативного отображения моделей данных и метода синтеза канонической модели были получены следующие результаты [1, 20].

Были построены аксиоматические расширения реляционной МД, эквивалентные разнообразным исходным структурным МД (сетевым, иерархическим, бинарным, дескрипторным). На рисунке 1.2 изображен пример аксиоматического расширения реляционной МД, эквивалентного сетевой модели данных КОДАСИЛ [15]. Семантика типов наборов модели КОДАСИЛ с MANDATORY и OPTIONAL классами членства в наборах выражается полными и частичными функциональными зависимостями, изображаемыми посредством составных аксиом.

На рисунках 1.3, 1.4, 1.5 показано множество средств канонической модели, полученной в результате ее синтеза. Применение метода отображения структурных моделей было осуществлено для случая, когда в качестве ядра канонической модели использовалась комбинация реляционной и слабоструктурированной модели. В качестве исходных моделей были использованы двенадцать широко распространенных различных структурных моделей данных начала 80х годов. Среди них присутствовали сетевые модели данных (включая КОДАСИЛ), иерархические модели данных (включая IMS), бинарные реляционные модели данных, слабоструктурированные модели (BASIS). Были построены требуемые отображения моделей данных, а также произведен синтез канонической модели.

Важное свойство процесса синтеза состоит в относительно быстром насыщении канонической модели, когда рассмотрение новых исходных моделей данных не привносит новых аксиом на уровне целевой МД. Полученная модель является насыщенной по отношению к

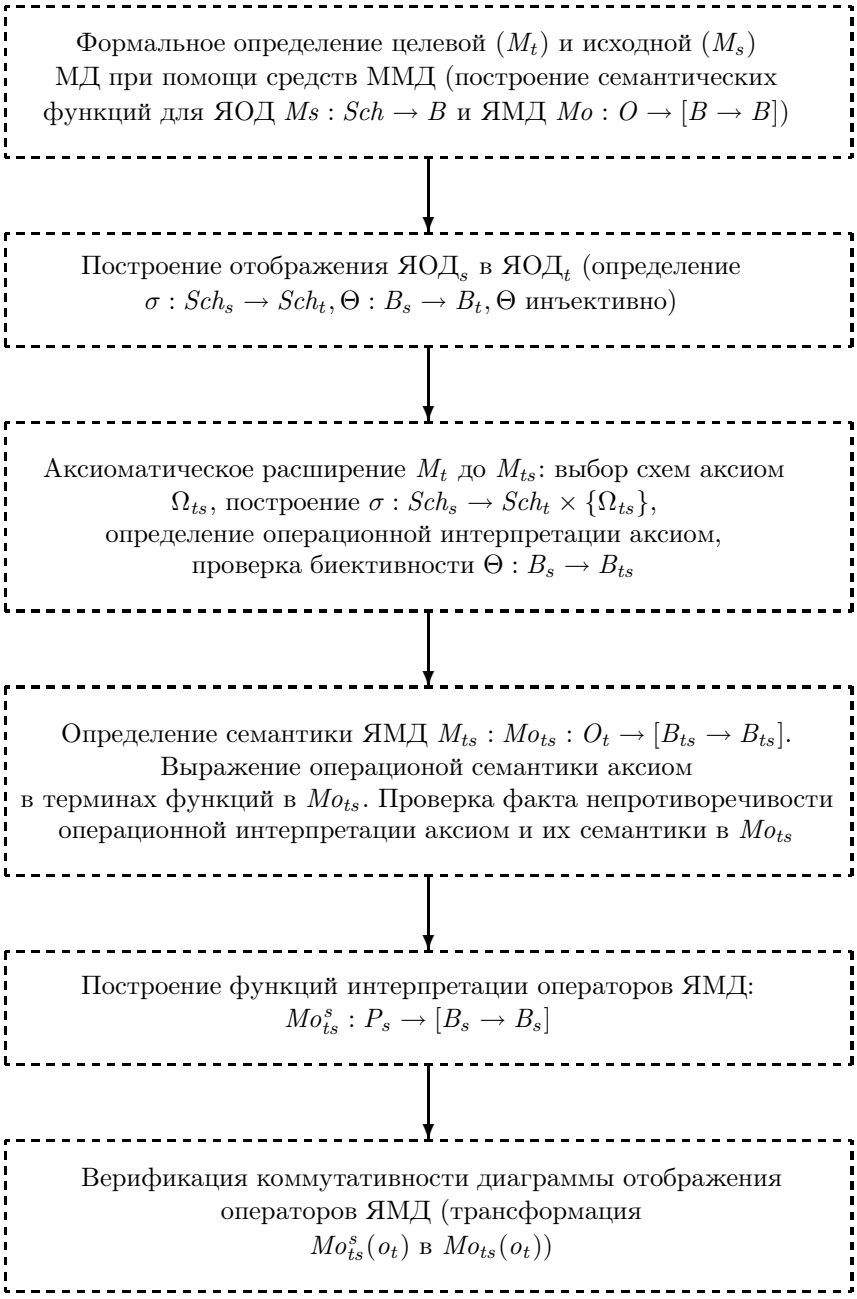


Рис. 1.1. Построение коммутативного отображения

известным на начало 80х годов моделям. Это обстоятельство позволяет считать результирующую модель канонической.

Простые аксиомы (для отношения R_i; A_i, A_j, A_k- наборы атрибутов R_i)

1.Аксиома уникальности
UNIQUE A_i
2.Аксиома постоянства
CONSTANT A_i
3.Аксиома определенности
OBLIGATORY A_i
4.Аксиома условной уникальности
UNIQUE NONNULL A_i
5.Аксиома порядка
R_i[RESTRICTED BY A_i] IS ORDERED[<order>]
[BY < direction > A_j{, < direction > A_k }]

Составные аксиомы (для отношений R_i, R_j)

6.Аксиома полной функциональной зависимости
$R_j(A_j) \rightarrow R_i(A_i)$
7.Аксиома частичной функциональной зависимости
$R_j(A_i) \Longrightarrow R_i(A_i)$
8.Аксиома частичной сильной функциональной зависимости
$R_i(A_i) = S \Longrightarrow R_i(A_i)$
9.Аксиома частичной функциональной зависимости с первоначальной связью
$R_j(A_j) = L \Longrightarrow R_i(A_i)$

Рис. 1.2. Аксиоматическое расширение КОДАСИЛ

Важно заметить, что для денотационной семантики, как для ММД, не существует инструментов проверки коммутативности диаграмм отображения ЯОД и ЯМД. Доказательство коммутативности диаграмм отображения ЯОД основывалось на методе структурной индукции. Доказательство коммутативности диаграмм отображения ЯМД основывалось на правилах эквивалентного преобразования функций метамодели. В связи с отсутствием специальных инструментов, построение отображения моделей в соответствии с этим подходом являлось

Средства канонической модели	Модели данных											
	1	2	3	4	5	6	7	8	9	10	11	12
Нормализованные отношения	*	*	*	*	*	*	*	*	*	*	*	*
Иерархические отношения		*				*	*	*		*		*
Позиционные аггрегаты							*	*				

Рис. 1.3. Ядро канонической модели данных

1.Реляционная МД Кодда (1970)
2.Сетевая МД КОДАСИЛ
3.Иерархическая МД IMS
4.Сетевая МД IDS
5.МД СУБД PALMA
6.МД ADABAS
7.Дескрипторная МД СУБД BASIS
8.МД СУБД ПОИСК
9.Сетевая МД TOTAL
10.Иерархическая МД СУБД INES
11.Бинарная реляционная МД
12.Реляционная МД Кодда (1979)

Рис. 1.4. Обозначения моделей данных

непростым. Вместе с тем, при сравнительно небольшом числе исходных моделей в среде, подход, основанный на построении отображения моделей и проверке их коммутативности вручную, является практически применимым, хотя и трудоемким.

Результаты, полученные для отображения структурных моделей, не теряют своего методологического значения и сегодня. Аналогичные подходы могут быть использованы для достижения других целей, например, получения общего представления корпоративных информационных моделей, отображения моделей в MDA.

В следующем разделе рассматривается более продвинутый метод автоматизированного доказательства корректности отображения моделей представления информации.

Средства канонической модели (аксиомы)	Модели данных											
	1	2	3	4	5	6	7	8	9	10	11	12
Уникальности		*	*	*	*				*		*	*
Постоянства		*	*	*	*				*			*
Определенности		*	*	*	*				*		*	*
Условной уникальности		*										
Условного постоянства		*										
Функция											*	
Частичная функция											*	
Порядка		*	*	*		*	*		*	*		
Предикат											*	
Полной ф.з.		*	*	*					*	*	*	*
Частичной ф.з.		*									*	*
Сильной ч.ф.з.		*										
Ч.ф.з. с первоначальной связью		*										
Полной ф.з. с обратной связью			*									
Ч.ф.з. с обратной связью											*	
Устойчивой полной ф.з.			*									
Устойчивой полной ф.з. с обратной связью			*									
Дуплексной зависимости					*	*						

Рис. 1.5. Расширения ядра

2 Метод коммутативного отображения объектных моделей

Период 90х характеризуется активным развитием и использованием объектных моделей и архитектур программного обеспечения промежуточного слоя, предназначенных для разработки интероперабельных информационных систем (например, [28]). В это время наряду с новыми объектными моделями (по существу являющимися расширяемыми), возникли формальные языки и методы разработки программ (основанные на исчислении уточнения и технике пошаговго уточнения). Это послужило предпосылкой развития методов отображения моделей и синтеза канонических моделей, определенных в предыдущем разделе. Вместо денотационной семантики была применена Нотация Абстрактных Машин (AMN) в качестве формальной метамодели данных. AMN обеспечивает манипулирование теоретико-множественными спецификациями в логике первого порядка и доказательство уточнения спецификаций [6, 7]. Техника уточнения позволила расширить основные определения отношений между типами данных, схемами, моделями данных так, чтобы вместо эквивалентности соответствующих спецификаций можно было рассуждать об их уточнении [7]. Специальные инструментальные средства (B-технология [6]) предоставляют возможность доказательства коммутативности диаграмм отображения моделей полуавтоматическим способом: теоремы, требуемые для доказательства уточнения моделей, генерируются B автоматически, но их доказательство может требовать участия человека.

2.1 AMN как объектная метамодель

AMN, как теоретико-модельная нотация, позволяет рассматривать интегрированно спецификацию пространства состояний и поведения (определенного операциями на состояниях). Спецификация состояния машины вводится переменными состояния вместе с инвариантами – ограничениями, которые должны всегда удовлетворяться. Операции определяются на основе расширения формализма охраняемых команд Дейкстры.

Ключевым понятием AMN является *уточнение*, позволяющее соотносить спецификации систем различных уровней абстракции. Уточняющая спецификация может быть значительно более детальной, чем уточняемая спецификация. Конструируется уточняющая спецификация на основе алгоритмического уточнения и уточнения данных [6]. Уточнение формализуется в AMN путем формулировки ряда теорем

специального вида, так называемых *proof obligations*. Такие теоремы формулируются автоматически при помощи инструментальных средств поддержки B-технологии (B-Toolkit, AntelierB) на основании *склеивающих инвариантов* – инвариантов, соотносящих состояния уточняемой и уточняющей системы. Теоремы могут быть доказаны при помощи инструментальных средств поддержки автоматического и (или) интерактивного доказательства.

2.1.1 Спецификация состояний системы в AMN

Язык спецификации состояний основан на теории множеств Цермело-Френкеля с аксиомой выбора и типизированном языке первого порядка со встроенными типами и конструкторами типов (сортов). Множество конструкторов сложных сортов включает: декартово произведение $(s \times t)$, множество всех подмножеств $(\mathbb{P}(s))$, выделение множества $(\{x \mid x \in s \wedge P(x)\})$, пересечение множеств $(s \cap t)$, объединение множеств $(s \cup t)$, разность множеств $(s - t)$, конструкторы реляционных сортов $(s \leftrightarrow t)$, конструкторы функциональных сортов $(s \longrightarrow t)$. Конструктор упорядоченной пары обозначается как (x, y) либо как $x \mapsto y$. Здесь s, t – множества, P – предикат.

Отношение между двумя множествами s и t представляет собой элемент множества $\mathbb{P}(s \times t)$, отношение r имеет область определения $\mathrm{dom}\,(r)$ и область значений $\mathrm{ran}\,(r)$. Функция f из множества s в множество t есть такое отношение, что каждому элементу $\mathrm{dom}\,(f) \subseteq s$ соответствует ровно один элемент множества t. Функция может быть частичной $f \in s \nrightarrow t$ или полной $f \in s \rightarrow t$. Для функции f и элемента $x \in \mathrm{dom}\,(f)$ определен терм $f(x)$, и его значение y $(y = f(x))$ такое, что $x \mapsto y \in f$.

В таблице 2.1 собраны основные теоретико-множественные обозначения AMN. Здесь S, T, U, s, t обозначают множества, причем $s \subseteq S$ и $t \subseteq T$; r, r_1, r_2, p, q – отношения из S в T, q – отношение из T в U.

Язык предикатов AMN сочетает теорию множеств и исчисление предикатов первого порядка (с обычным набором связок $\wedge, \vee, \Rightarrow, \neg$, кванторами существования $\exists$ и всебщности $\forall$ и предикатом принадлежности $\in$) с простой арифметической теорией. Интерпретация состояния абстрактной машины осуществляется переменными машины - каждой переменной присваивается элемент из определенного домена.

2.1.2 Спецификация операций в AMN

Операции абстрактных машин основаны на обобщенных подстановках. Любая операция в AMN имеет следующий вид.

Обозначение AMN	Значение обозначения
$S \leftrightarrow T$	$\mathbb{P}(S \times T)$
$\mathrm{dom}\,(r)$	$\{x \mid x \in S \wedge \exists\, y \bullet (x \in T \wedge x \mapsto y \in r)\}$
$\mathrm{ran}\,(r)$	$\{y \mid y \in T \wedge \exists\, x \bullet (x \in S \wedge x \mapsto y \in r)\}$
$p;\ q$	$\{x \mapsto z \mid x \mapsto z \in S \leftrightarrow U \wedge$ $\exists\, y \bullet (y \in T \wedge x \mapsto y \in p \wedge y \mapsto z \in q)\}$
$\mathrm{id}\,(S)$	$\{x \mapsto y \mid x \mapsto y \in S \times S \wedge x = y\}$
$s \lhd r$	$\{x \mapsto y \mid x \mapsto y \in r \wedge x \in s\}$
$s \rhd r$	$\{x \mapsto y \mid x \mapsto y \in r \wedge y \in t\}$
$s \lhd\!\!\!- r$	$\{x \mapsto y \mid x \mapsto y \in r \wedge x \in S - s\}$
$s \rhd\!\!\!- r$	$\{x \mapsto y \mid x \mapsto y \in r \wedge y \in T - t\}$
r^{-1}	$\{y \mapsto x \mid y \mapsto x \in T \times S \wedge x \mapsto y \in r\}$
$r[s]$	$\{y \mid y \in T \wedge \exists\, x \bullet (x \in s \wedge x \mapsto y \in r)\}$
$r_1 \lhd\!\!\!- r_2$	$(\mathrm{dom}\,(r_2) \lhd\!\!\!- r_1) \cup r_2$
$s \twoheadrightarrow t$	$\{r \mid r \in s \leftrightarrow t \wedge (r^{-1};\ r) \subseteq \mathrm{id}\,(t)\}$

Таблица 2.1. Теоретико-множественные обозначения AMN

$$r_1, \ldots, r_n \leftarrow op(p_1, \ldots, p_m) = S$$

Здесь op – имя операции, $r_1, \ldots, r_n$ – выходные параметры операции, $p_1, \ldots, p_m$ – выходные параметры операции, S – подстановка, определяющая действие операции на пространстве состояний.

Язык обобщенных подстановок (Generalized Substitution Language, GSL) позволяет описывать переходы между состояниями системы. Каждая обобщенная подстановка S определяет преобразователь предиката, связывающий некоторое постусловие R со своим *слабейшим* предусловием $[S]R$, что гарантирует сохранение R после выполнения операции. В таком случае говорят, что S *устанавливает* R. "Слабейшее" предусловие означает, что предикат "начального состояния", связанный с некоторым предикатом "заключительного состояния", должен разрешать максимально большое число состояний. В таблице 2.2 рассматриваются основные виды обобщенных подстановок и соответствующие им слабейшие предусловия. Здесь S, T, T_1, T_2, S_1, S_2 означают подстановки, x, y, t – переменные, E, F – выражения, G, G_1, G_2, H, P – предикаты, $P\{x \to E\}$ – предикат P, в котором все свободные вхождения переменной x заменены на E.

Обобщенная подстановка S	$[S]P$
$x := E$	$P\{x \rightarrow E\}$
skip	P
$x := E \mid\mid y := F$	$[x, y := E, F]P$
$S[]T$	$[S]P \wedge [T]P$
SELECT G_1 THEN T_1 WHEN G_2 THEN T_2 END	$(G_1 \Rightarrow [T_1]P) \wedge (G_2 \Rightarrow [T_2]P)$
PRE G THEN T END	$G \wedge [T]P$
ANY t WHERE G THEN T END	$\forall\, t \bullet (G \Rightarrow [T]P)$
$S\,;\ T$	$[S][T]P$
IF G THEN S ELSE T END	$(G \Rightarrow [S]P) \wedge (\neg\ G \Rightarrow [T]P)$

Таблица 2.2. Обобщенные подстановки и их семантика

2.1.3 Виды конструкций и структурные механизмы AMN

В AMN существует три вида конструкций:

- *абстрактная машина (abstract machine),*
- *уточнение(refinement),*
- *реализация (implementation).*

Абстрактная машина может быть только уточняемой конструкцией, и при описании операций абстрактной машины не разрешается использовать последовательную и циклическую подстановки. Реализация может быть только уточняемой конструкцией, при описании операций реализации не разрешается использовать недетерминированные подстановки (*SELECT, ANY*), параллельную подстановку и предусловие. Реализации также не разрешается иметь собственных переменных. Уточнение является более универсальной конструкцией, т.к. может использоваться как в качестве уточняемой, так и в качестве уточняющей конструкции, язык описания операций уточнения не имеет таких ограничений, как в абстрактной машине и реализации. Поэтому конструкция уточнения является наиболее предпочтительной для однородного представления в AMN спецификаций канонической модели.

Уточнение выглядит следующим образом.

REFINEMENT r

REFINES m
SEES sm
INCLUDES im
SETS s
CONSTANTS c
PROPERTIES $P(s, c)$
VARIABLES x
INVARIANT $I(x)$
INITIALISATION S
OPERATIONS O_1; ...; O_n
END

Уточнение с именем r содержит переменные в разделе *VARIABLES*, которые и определяют состояние системы. Начальная инициализация переменных определеяется подстановкой, определенной в разделе *INITIALISATION*. Изменять состояние системы могут только операции, определенные в разделе *OPERATIONS*. Состояние системы должно удовлетворять инварианту после инициализации; операции также должны сохранять инвариант. Раздел *SETS* содержит определение множеств, раздел *CONSTANTS* содержит имена констант, использующихся в уточнении, раздел *PROPERTIES* содержит предикат, описывающий свойства констант. Имя конструкции, которую уточняет r, содержится в разделе *REFINES*.

Разделы *SEES* и *INCLUDES* отвечают за композицию уточнения с другими конструкциями. Композиция *SEES* используется в случае, если нескольким конструкциям необходимо обеспечить возможность чтения значений переменных, множеств и констант некоторой машины. Композиция *INCLUDES* используется в случае, если конструкции необходимо использовать другую конструкцию как свою подсистему. Переменные включаемой конструкции становятся переменными включающей конструкции; их изменение может производиться только при помощи операций включаемой конструкции. Инвариант включаемой конструкции становится частью инварианта включающей конструкции.

2.1.4 Формализация понятия уточнения в AMN

Рассмотрим, каким образом формализуется факт уточнения конструкции M конструкцией N в AMN. Заметим, что факт уточнения может быть установлен только в том случае, если конструкции M и N (таблица 2.3) *согласованы*, т.е. удовлетворяют следующим требованиям.

- Раздел *REFINES* конструкции N должен содержать имя M.

- Для каждой операции конструкции M, конструкция N должна содержать операцию с точно такой же сигнатурой.

- Инвариант конструкции N должен содержать так называемый *склеивающий* инвариант (инвариант уточнения) R, задающий соотношение между состояниями уточняемой и уточняющей конструкций.

REFINEMENT M	REFINEMENT N
REFINES K	REFINES M
CONSTANTS c_M	CONSTANTS c_N
PROPERTIES P_M	PROPERTIES P_N
VARIABLES v	VARIABLES w
INVARIANT I_M	INVARIANT I_N
INITIALISATION $Init_M$	INITIALISATION $Init_N$
OPERATIONS	OPERATIONS
$y \leftarrow op(x) =$	$y \leftarrow op(x) =$
PRE $Pre_{op,M}$	PRE $Pre_{op,N}$
THEN	THEN
$Def_{op,M}$	$Def_{op,N}$
END	END
...	...
END	END

Таблица 2.3. Спецификации M и N

Определение 2.1 M **уточняет** N, *если верны следующие теоремы (proof obligations).*

- *Теорема* **непустоты объединенного состояния**. *Существует объединенное состояние M и N, удовлетворяющее инвариантам M и N.*

$$P_M \wedge P_N \Rightarrow \exists(v, w) \bullet (I_M \wedge I_N)$$

- *Теорема* **уточнения инициализации**. *Инициализация N уточняет инициализацию M.*

$$P_M \wedge P_N \Rightarrow [Init_N]\neg\, [Init_M]\neg\, I_N$$

- *Теорема* **уточнения операций**. *Каждая из операций M уточняет соответствующую операцию N, т.е. при условии выполнения инварианта уточнения и предусловия уточняемой операции, выполняется предусловие уточняющей операции; и для каждого случая исполнения $Def_{op,N}$, существует исполнение $Def_{op,M}$ из соответствующего начального состояния (задаваемого инвариантом уточнения R), которое устанавливает точно такие же значения выходных параметров и сохраняет инвариант уточнения на пост-состояниях.*

$$P_M \wedge P_N \wedge I_M \wedge I_N \wedge Pre_{op,M} \Rightarrow \\ Pre_{op,N} \wedge [Def_{op,N}\{y \rightarrow y'\}] \neg [Def_{op,M}] \neg (I_N \wedge y' = y)$$

2.2 Уточнение принципов отображения модели данных

Понятие *эквивалентности моделей данных* переопределяется как понятие *уточнения моделей данных*, понимаемое следующим образом. Два состояния базы данных - одно в исходной модели, другое в целевой модели - находятся в отношении уточнения, если они отображаются в два состояния *абстрактной метамодели данных* так, что образ состояния исходной модели является уточнением образа состояния целевой модели. Тип данных t_s биективно уточняет тип t_t если, и только если эти типы продуцируют множества состояний базы данных равной мощности, связанные биективно таким образом, что соответствующие состояния находятся в отношении уточнения. Схема S_s уточняет схему S_t если, и только если для каждого типа t_s в S_s есть тип t_t в S_t (и S_t не содержит других типов), такой, что t_s является уточнением t_t. Модель данных M_s *уточняет*, модель данных M_t если, и только если для каждой допустимой схемы S_s в M_s существует допустимая схема S_t в M_t такая, что S_s является уточнением S_t.

Принцип аксиоматического расширения моделей данных переопределяется как принцип аксиоматического расширения типов данных, так, что биективное отображение θ в коммутативной диаграмме отображения состояний типов данных (которая заменяет диаграмму отображения схем) становится отношением уточнения данных вместо отношения эквивалентности данных. Вместо диаграммы отображения операторов языка манипулирования данными, используется коммутативная диаграмма поведения типов данных, в которой π становится алгоритмическим уточнением.

В целом, для отображения модели данных необходимо построить

1) отображение M_j в расширение M_i; 2) AMN семантику для M_j; 3) AMN семантику для расширенной M_i. После этого применяется B технология, чтобы доказать a) свойства отображения, основанные на состояниях (коммутативность диаграмм состояния типов данных); b) поведенческие свойства отображения для всех моделей типов, определенных для некоторой исходной модели данных. Это ведет к доказательству того, что M_j является уточнением расширения M_i. Важно, что отображение моделей данных, основанное на AMN и на технике уточнения, применимо как к структурированным, так и к объектным моделям данных. Пример отображения типа связи ODMG'93 [27] в метатип ассоциации СИНТЕЗа [2] на основе техники уточнения содержится в [21].

Метатип ассоциации СИНТЕЗа - это свободный тип без ограничений, который должен быть правильно расширен так, чтобы он мог быть уточнен типом связи ODL (конкретным, встроенным типом). Уточнение данных диаграммы состояния типов данных достигается путем введения адекватных аксиом для метатипа ассоциации (аксиома частичной функциональной зависимости и аксиома порядка). Отображение операций обеспечивается спецификацией операций связи в метатипе ассоциации: add_one_to_one, remove_one_to_one, create, delete, и traverse. Были определены абстрактная интерпретация метатипа ассоциации расширенного СИНТЕЗа в AMN и машина, соответствующая типу связи ODMG, как уточнение машины метатипа ассоциации; после этого было доказано уточнение ассоциации.

2.3 Общие особенности подхода к коммутативному отображению моделей данных

Таким образом, для создания необходимых оснований разработки семантически интероперабельных неоднородных сред моделирования данных, используются следующие принципы: принцип расширения моделей данных, принцип коммутативного отображения моделей данных, принцип синтеза унифицирующей канонической модели данных, основанный на понятии *уточнения моделей данных*.

До сих пор методы разработки расширяемой канонической модели и коммутативные отображения рассматривались для структурированных и объектно-ориентированных моделей. Далее их действие будет распространено на процессные модели. Класс процессных моделей представляет собой важный класс моделей в контексте информационных систем различного назначения. Так, например, модели корпораций выражают при помощи действий, заданных в виде спецификаций одновременно выполняемых процессов. Спецификации виртуальных

корпораций основаны на интегрированных процессах реальных корпораций, участвующих в процессе интеграции. Процессы реализуются как потоки работ. Технология потоков работ продолжает разрабатываться как для традиционных приложений моделирования и координации процессов в организациях, так и для компонентно ориентированных подходов проектирования потоков работ.

Выше было показано, как сохранить информацию и операции при отображении одной модели данных в другую. При отображении процессных моделей также должно быть сохранено поведение одновременно выполняемых процессов.

Процессы относятся к классу интерактивных вычислений [37, 38]. Понятие интерактивных вычислений характеризуется недетерминизмом (выбор проявляемого поведения осуществляется под влиянием среды), одновременностью (рассматриваемой для композиции взаимодействующих одновременных процессов), взаимодействием между компонентами и внешней средой, а также ограничениями, накладываемыми средой на вычисления.

Классическая «эффективная вычислимость» (в рамках которой мы оставались в предыдущих разделах) является чисто алгоритмической, в то время как парадигма интерактивных вычислений имеет дело с одновременными и распределенными вычислениями, которые координируются протоколами. Широко известны недетерминированные системы одновременных процессов, основанные на CCS Р.Милнера [26], процессных алгебрах [10] и других формализмах, которые были разработаны для изучения взаимодействия одновременных процессов. В то же время ни одна из существующих моделей не могла служить расширяемым ядром канонической модели процессов в силу отсутствия общей теории одновременного поведения. В следующем разделе будут рассмотрены необходимые приемы создания такой модели.

3 Построение канонической модели процессов

3.1 Разнообразие процессных моделей

Процессные модели активно разрабатываются в мире в течение нескольких десятилетий для практических целей и целей исследований. Согласно весьма общей классификации, процессные модели подразделяются на интенсиональные модели (LTS, сети Петри и ряд других моделей, имитирующих процессы состояниями и переходами между ними), экстенсиональные модели (CSP, CCS, ACP и другие процессные ал-

гебры, имитирующие системы в терминах внешнего наблюдателя как трассы событий, разнообразные деревья синхронизации, событийные структуры, и пр.), а также модели, имитирующие одновременные процессы чередованием событий (в этот класс попадают процессные алгебры) либо подлинно одновременными действиями (например, трассированием событий, экстенсиональными моделями синхронизации, и пр.). Важное место в этом многообразии занимают сети Петри, которые являются интенсиональными моделями с имитацией подлинно одновременных действий.

Наиболее широко практическое использование процессные модели находят в разнообразных Системах Управления Потоками Работ (СУПР), обеспечивающих описание процессов, которые необходимо поддерживать в информационных системах, и управление ими. Процессные модели СУПР обычно не формализованы и являются чрезвычайно разнообразными (их число измеряется многими десятками и определяется большим числом независимых компаний – разработчиков СУПР). Вместе с тем, реальные задачи композиционного проектирования распределенных неоднородных систем приходится решать, опираясь как раз на такие прагматические модели СУПР. При этом пригодные для конкретной информационной системы фрагменты существующих процессов (выраженные в неоднородных моделях потоков работ разнообразных СУПР) должны уточнять фрагменты спецификаций процессов проектируемой системы, выраженные в канонической процессной модели. Учитывая многообразие процессных моделей СУПР, обеспечение полноты их охвата синтезированной канонической моделью процессов является непростой задачей.

3.2 Полнота набора процессных моделей для синтеза канонической модели процессов

Недавно группой ученых во главе с Ван дер Аальстом [4] была выполнена работа по анализу процессных моделей систем управления потоками работ (влючая Staffware, COSA, InConcert, Eastman, FLOWer, Domino, Meteor, Mobile, MQSeries, Forte, Verve, Vis. WF, Changeng., I-Flow, SAP/R3), а также языков процессной композиции Web сервисов (XPDL, UML, BPEL4WS, BPML, XLANG, WSFL, WSCI).

При этом был выполнен анализ типовых конструкций в рассмотренных процессных моделях. Этот анализ позволил выявить полный набор образцов процессных моделей потоков работ (workflow patterns). Определено 20 образцов, которые разбиты на следующие группы в соответствии со сложностью и типом используемых конструкций:

1. Основные образцы потока управления

2. Образцы сложной синхронизации и ветвления
3. Образцы структурирования
4. Образцы, включающие множественные экземпляры
5. Образцы, использующие концепцию состояния
6. Образцы прекращения деятельностей

Анализ процессных моделей потоков работ разнообразных СУПР показал, что введенных образцов достаточно для моделирования их конструкций (более того, существующие СУПР реализуют лишь различные слабые подмножества полного набора образцов). С другой стороны, полнота этого набора образцов подтверждается тем, что введенного разнообразия достаточно для описания произвольных реальных потоков работ.

Эти результаты позволяют выбрать для синтеза канонической модели процессов в качестве исходных процессных моделей указанные образцы потоков работ. Тем самым удастся сочетать процессную полноту выбранного набора конструкций с возможностью моделирования этим набором произвольных процессов, выражаемых процессными моделями разнообразных СУПР.

3.3 Определение ядра канонической модели процессов

3.3.1 Требования и мотивация

Каноническая процессная модель разрабатывается в виде ядра, описывающего основополагающие примитивы спецификаций процессов, и его расширений. Один из основных принципов отображения некоторой процессной модели в каноническую заключается в том, что отображение осуществляется в расширение ее ядра, определяемое так, чтобы расширенная модель уточнялась исходной моделью. Синтез канонической модели реализуется так, что фиксируется ее ядро, расширения которого конструируются так, чтобы их уточняли всевозможные исходные процессные модели (в нашем случае образцы потоков работ), после чего объединение всех таких расширений вместе с ядром составит результат синтеза канонической модели.

Синтез расширяемой канонической процессной модели предполагается реализовать на основе формальной системы (Нотации Абстрактных Машин (AMN)), позволяющей определять теоретико-модельные спецификации в логике первого порядка и осуществлять доказательство факта уточнения спецификаций. Теория уточнений позволяет

развить определения отношений между состояниями, типами данных, переходами из состояния в состояние, функциями так, чтобы вместо эквивалентности соответствующих спецификаций, можно было доказательно рассуждать об их уточнении.

В качестве ядра канонической процессной модели было выбрано подмножество языка скриптов СИНТЕЗа, наиболее близкое к раскрашенным сетям Петри согласно [18]. Ядро обладает следующими свойствами.

1. Основано на хорошо изученной и распространенной модели сетей Петри.

2. Вводит сети Петри в объектную среду. В результате потоки управления (control flow) и потоки данных (data flow) удается соединить, используя токены - объекты, принадлежащие определенному типу и обладающие уникальными идентификаторами.

3. Предоставляет средства связывания переходов сетей Петри с функциями, которые должны быть выполнены при срабатывании этих переходов. При этом также задаются правила связывания входных и выходных токенов перехода с входными и выходными параметрами функции. Такое связывание необходимо для моделирования информационных систем в целом, что не принимается во внимание в более абстрактных моделях [18, 5].

3.3.2 Синтаксис ядра канонической модели потоков работ

В канонической модели процессы описывается с помощью скриптов.

Определение любой сущности в языке СИНТЕЗ (например, типы, классы, функции) дается с помощью *фрейма*. Фрейм может рассматриваться как структурированная символическая модель некоторой сущности или понятия, используемая для представления их отдельных экземпляров. Синтаксически фрейм представляется в фигурных скобках. Имена слотов и их значения разделяются двоеточием. Значения слотов разделяются запятыми. В качестве значений слотов могут выступать атомарные величины, фреймы, коллекции формул исчисления объектов, а также множества значений. Различные слоты разделяются точкой с запятой.

Скрипты определяются с помощью родового типа скрипта в языке СИНТЕЗ и могут образовывать иерархию с отношением тип-подтип. Родовой тип – это наименее конкретизированный тип. Ниже приводится определение родового типа скрипта в форме Бэкуса-Наура. Каждый экземпляр типа скрипта отвечает некоторому исполнению потока работ, определяемому данным скриптом.

```
<script type identifier> ::=
{ <type identifier>; in: script;
  [params: {<formal parameter list>};]
  [supertype: {<supertype name list>};]
  instance_section:
  {
   [<attribute specification list>;]
   [initial: <initial marking>;]
   states: <state section>;
   [transitions: <transition section>;]
  }
}
```

Здесь `<type identifier>` – имя типа скрипта. В необязательном слоте params задаются формальные параметры типа. Супертипы данного типа (если они есть) перечисляются в слоте `supertype`. Экземпляр типа определяет слот `instance_section`. Для каждого экземпляра в слоте `initial` определяется начальная разметка – множество пар (имя состояния, константа). Константа задает значение токена, находящегося в начальный момент в указанном состоянии.

```
<initial marking> ::=
 <element of initial marking list> |
 <element of initial marking list>, <initial marking>
<element of initial marking list> ::=
 {<state identifier> , <constant>}
```

Слот `states` определяет множество мест конкретной сети. Каждое место характеризуется именем и типом токенов, которые могут в нем находиться.

```
<state section> ::=
 <state specification> |
 <state specification>, <state section>
<state specification> ::=
 {
  <state name>;
  token: <type>;
 }
```

В слоте `transitions` определяется множество переходов конкретной сети.

```
<transition section> ::=
 <transition specification> |
 <transition specification>, <transition section>
```

Каждый переход характеризуется именем (`<transition name>`), списком входных мест (`from`), списком выходных мест (`to`), списком условий (`conditions`), функцией, которая выполняется при срабатывании перехода (`activity`), а также списками связывания входных и выходных параметров этой функции (`bind_from`, `bind_to`).

```
<transition specification> ::=
 {
  <transition name>;
  from: <list of input state names>;
  [bind_from: <binding list>;]
  to: <list of output state names>;
  [bind_to: <binding list>;]
  [conditions: <assertion list>;]
  activity: <function declaration>
 }
```

В слоте `from` определяются места, из которых токены могут входить в данный переход при его срабатывании. В слоте `to` определяются места, в которых могут появляться токены при срабатывании данного перехода. Слот `conditions` определяет условия, при которых переход может сработать (помимо наличия необходимого числа токенов определенных типов в определенных местах).

Слот `activity` задает функцию, выполняемую при срабатывании перехода. При этом входными параметрами этой функции являются токены, входящие из мест, описанных в секции `bind_from`, а выходными – токены, помещаемые в места, описанные в секции `bind_to`. В этих двух секциях также задаются условия, в соответствии с которыми токены могут перемещаться.

```
<binding list> ::=
 <element of binding list> |
 <element of binding list>, <binding list>
<element of binding list> ::=
 { <state identifier>,
   <input/output parameter name> [, <condition>] }
```

3.4 Формальная семантика ядра канонической модели процессов

Скрипты позволяют описывать процессы как последовательности действий, каждое из которых может вызывать смену состояний ИС, задаваемой спецификацией канонической модели. Базой для определения отображения скриптов в AMN является отображение абстрактных типов данных канонической модели (языка СИНТЕЗ) в AMN [3, 13], а также ряд исследований по моделированию процессных моделей в AMN.

Батлером [14] был разработан язык, сочетающий ограниченное подмножество процессной алгебры CSP [17] и AMN, а также методы и средства отображения комбинированного языка в AMN. CSP при этом использовался как язык структуризации абстрактных машин. Естественным расширением данного подхода стала работа [33], расширяющая метод отображения подмножества CSP в AMN до отображения полного Timed CSP [30](CSP, расширенный временными примитивами) в AMN. Идея комбинации CSP и AMN также разрабатывалась в работе Шнайдера [31], где подмножество CSP использовалось для управления абстрактными машинами AMN.

В подходах к интеграции UML и AMN, Батлер [32] и Леданг [24] используют идею охраняемых (guarded) подстановок для моделирования событий в диаграммах состояний (state-charts) UML.

Наконец, Абриаль в своих последних работах [8] по расширению AMN для моделирования взаимодействующих систем вводит понятие события, действующего на глобальном пространстве состояний системы и не имеющего параметров. Действие события на пространстве состояний по Абриалю также выражается охраняемой подстановкой.

Благодаря всем перечисленным работам сформировалось общее понимание, каким образом следует моделировать процессные модели в AMN. Это понимание, в свою очередь, позволило разработать метод отображения модели скриптов в AMN.

Основная идея отображения скриптов в AMN заключается в моделировании состояния системы (т.е. мест скрипта) при помощи переменных AMN, и моделировании переходов при помощи операций AMN, тела которых представляют из себя охраняемые подстановки. Такая идея характерна для всех подходов, применяемых для представления процессных моделей в AMN.

3.4.1 Общие принципы определения семантики скриптов

Рассмотрим тип скрипта S, заданный следующей спецификацией.

```
{ S; in: script;
  params: { pf/function, PT/type };

  instance_section:
  {
    states: ... ;
    initial: ... ;
    transitions: ... ;
  }
}
```

Скрипт S представляется в AMN конструкцией *REFINEMENT* с именем S. Использование конструкции *REFINEMENT*, а не конструкций *MACHINE* или *IMPLEMENTATION* обусловлено необходимостью представления скриптов в AMN однородными структурами, поскольку любой скрипт может выступать как в роли уточняемой, так и в роли уточняющей спецификации.

Композиция машины S с машинами, представляющими другие абстрактные типы, производится следующим образом. Рассмотрим абстрактный тип S_F, спецификацию которого составляют методы, задаваемые функциями переходов скрипта. Машина S состоит ровно в тех отношениях композиции с машинами, представляющими другие абстрактные типы, в которых состояла бы с этими машинами машина S_F, представляющая тип S_F в AMN. Аналогично машинам, представляющим абстрактные типы, машина S состоит в отношении *SEES* с контекстной машиной модуля M *ContextM*.

Каждое из мест скрипта, описанное в слоте `states`, представляется в AMN отдельной переменной. Подробно данное представление описывается в разделе 3.4.2.

Каждый из переходов скрипта, описанный в слоте `transitions`, представляется в AMN операцией машины S. Подробно данное представление описывается в разделе 3.4.3.

3.4.2 Представление мест скрипта в AMN

В случае, если скрипт является родовым, и имеет параметр-тип PT, экстенсионал параметра определяется константами ext_PT, $extp\_PT$ в разделе *PROPERTIES* следующим образом.

$$ext\_PT \in \mathbb{P}(Obj) \wedge extp\_PT \in \mathbb{P}(Obj) \wedge extp\_PT \subseteq ext\_PT$$

Если скрипт имеет параметр-функцию pf, то считается, что параметр-функция имеет минимально необходимое количество входных и вы-

ходных параметров, определяемое сигнатурой функции слота `activity` перехода.

Каждое из мест скрипта, описанное в слоте `states`, представляется в AMN отдельной переменной. Место

```
{ s; token: T; }
```

где T - абстрактный тип, представляется переменной s, типизированной в разделе *INVARIANT* машины как

$$s \in \mathbb{P}(ext\_T)$$

В случае, если начальная маркировка места s не описана в слоте `initial` скрипта, в разделе *INITIALISATION* машины переменная s инициализируется подстановкой

$$s := \varnothing$$

Если начальная маркировка места s описана элементом списка маркировки

$$\{s, c\}$$

где c - константа, то переменная s инициализируется подстановкой

$$s := mTerm[\![c]\!]$$

где $mTerm$ - семантическая функция отображения в AMN термов канонической модели.

3.4.3 Представление переходов скрипта операциями AMN

Рассмотрим переход, задаваемый следующей спецификацией.

```
{ t;
  from:
   s11, s12,
   s21, s22;
  bind_from:
   {s11, p1, b11}, {s12, p1, b12},
   {s21, p2, b21}, {s22, p2, b22};
  to: s31, s32;
  bind_to:
   {s31, r1, b31}, {s32, r1, b32},
   {s41, r2, b41}, {s42, r2, b42};
  conditions: C1, ... , Cn;
```

```
  activity: { in: function;
   params: { +p1/T1, +p2/T2, -r1/T3, -r2/T4, -o/T5 };
   { predicative: { f } }
  }
}
```

Функция перехода имеет два входных параметра p_1, p_2 и три выходных параметра r_1, r_2, o. Места s_1^1, s_2^1 с типом токенов T_1 связываются с параметром p_1 условиями b_1^1, b_2^1, соответственнно. Места s_1^2, s_2^2 с типом токенов T_2 связываются с параметром p_2 условиями b_1^2, b_2^2, соответственнно. Места s_1^3, s_2^3 с типом токенов T_3 связываются с параметром r_1 условиями b_1^3, b_2^3, соответственнно. Места s_1^4, s_2^4 с типом токенов T_4 связываются с параметром r_2 условиями b_1^4, b_2^4, соответственнно. С параметром o не связываются никакие места.

Переход t представляется в AMN следующей операцией.

$r_1, r_2, o \leftarrow t(p_1, p_2) =$
SELECT
  $mPredicate[\![C_1]\!] \wedge \ldots \wedge mPredicate[\![C_1]\!] \wedge$
  $\exists\, t_1 \bullet (t_1 \in s_1^1 \wedge mPredicate[\![b_1^1]\!]) \vee$
  $\exists\, t_1 \bullet (t_1 \in s_2^1 \wedge mPredicate[\![b_2^1]\!]) \wedge$
  $\exists\, t_2 \bullet (t_2 \in s_1^2 \wedge mPredicate[\![b_1^2]\!]) \vee$
  $\exists\, t_2 \bullet (t_2 \in s_2^2 \wedge mPredicate[\![b_2^2]\!])$
THEN
  ANY t_1, t_2 WHERE
    $t_1 \in Obj \wedge t_2 \in Obj \wedge$
    $(t_1 \in s_1^1 \wedge mPredicate[\![b_1^1]\!] \vee t_1 \in s_2^1 \wedge mPredicate[\![b_2^1]\!]) \wedge$
    $(t_2 \in s_1^2 \wedge mPredicate[\![b_1^2]\!] \vee t_2 \in s_2^2 \wedge mPredicate[\![b_2^2]\!])$
  THEN
    $s_1^1 := s_1^1 - \{t_1\} \,||\, s_2^1 := s_2^1 - \{t_1\} \,||$
    $s_1^2 := s_1^2 - \{t_2\} \,||\, s_2^2 := s_2^2 - \{t_2\} \,||$
     BEGIN
       $mSubstitution[\![f]\!]\{p_1 \mapsto t_1, p_2 \mapsto t_2\}$;
       (SELECT $mPredicate[\![b_1^3]\!]$ THEN
         $s_1^3 := s_1^3 \cup \{r_1\}$
        WHEN $mPredicate[\![b_2^3]\!]$ THEN
         $s_2^3 := s_2^3 \cup \{r_1\}$
        END ||
        SELECT $mPredicate[\![b_1^4]\!]$ THEN
         $s_1^4 := s_1^4 \cup \{r_2\}$
        WHEN $mPredicate[\![b_2^4]\!]$ THEN
         $s_2^4 := s_2^4 \cup \{r_2\}$

```
      END )
    END
  END
END
```

Тело операции представляет собой подстановку *SELECT*, условным предикатом которой является условие срабытывания перехода, извлекаемое из связываний слота `bind_from`. Представление условий связывания в AMN определяется семантической функцией *mPredicate*, преобразующей формулы канонической модели в предикаты AMN.

В случае выполнения условия срабатывания перехода, из мест, определенных связываниями слота `bind_from`, выбираются подходящие токены для передачи в качестве входных параметров в функцию перехода. Выбранные токены удаляются из своих мест.

Одновременно с этим выполняется подстановка, представляющая собой последовательную композицию двух подстановок, первая из которых представляет функцию перехода, а вторая представляет правильное размещение токена, полученного на выходе перехода.

Тело подстановки, представляющей функцию перехода, формирует семантическая функция *mSubstitution*; в полученном теле имена входных параметров p_1, p_2 заменяются на имена переменных t_1 и t_2, хранящих выбранные токены.

Токен, полученный в результате выполнения подстановки, представляющей функцию перехода, размещается в одном из мест, определенном связываниями слота `bind_to` при помощи охраняемой подстановки *SELECT*.

Действие перехода на пространстве состояний системы может задаваться функцией-параметром *pf* скрипта. При этом формула f имеет вид вызова функции-параметра $pf(p_1, p_2, r_1, r_2, o)$ с соответствующими фактическими параметрами. Действие функции-параметра на пространстве состояний системы моделируется произвольным выбором выходных параметров.

```
ANY r1', r2', o'
WHERE r1' ∈ ext_T3 ∧ r2' ∈ ext_T4 ∧ o' ∈ ext_T5
THEN
  r1 := r1' ||
  r2 := r2' ||
  o := o1
END
```

В случае, если скрипт специфицирует лишь поток управления, т.е. не конкретизирует действие `activity` на пространстве состояний системы, вид операции, соответствующей переходу, упрощается.

$t =$
SELECT
  $mPredicate[\![C_1]\!] \wedge \ldots \wedge mPredicate[\![C_1]\!] \wedge$
  $\exists\, t_1 \bullet (t_1 \in s_1^1 \wedge mPredicate[\![b_1^1]\!]) \vee$
  $\exists\, t_1 \bullet (t_1 \in s_2^1 \wedge mPredicate[\![b_2^1]\!]) \wedge$
  $\exists\, t_2 \bullet (t_2 \in s_1^2 \wedge mPredicate[\![b_1^2]\!]) \vee$
  $\exists\, t_2 \bullet (t_2 \in s_2^2 \wedge mPredicate[\![b_2^2]\!])$
THEN
  ANY t_1, t_2 WHERE
    $t_1 \in Obj \wedge t_2 \in Obj \wedge$
    $(t_1 \in s_1^1 \wedge mPredicate[\![b_1^1]\!] \vee t_1 \in s_2^1 \wedge mPredicate[\![b_2^1]\!]) \wedge$
    $(t_2 \in s_1^2 \wedge mPredicate[\![b_1^2]\!] \vee t_2 \in s_2^2 \wedge mPredicate[\![b_2^2]\!])$
  THEN
    $s_1^1 := s_1^1 - \{t_1\} \;||\; s_2^1 := s_2^1 - \{t_1\} \;||$
    $s_1^2 := s_1^2 - \{t_2\} \;||\; s_2^2 := s_2^2 - \{t_2\} \;||$
     ANY r_1', r_2', o'
     WHERE $r_1' \in ext\_T_3 \wedge r_2' \in ext\_T_4 \wedge o' \in ext\_T_5$
     THEN
       SELECT $mPredicate[\![b_1^3]\!]$ THEN
         $s_1^3 := s_1^3 \cup \{r_1\}$
       WHEN $mPredicate[\![b_2^3]\!]$ THEN
         $s_2^3 := s_2^3 \cup \{r_1\}$
       END ||
       SELECT $mPredicate[\![b_1^4]\!]$ THEN
         $s_1^4 := s_1^4 \cup \{r_2\}$
       WHEN $mPredicate[\![b_2^4]\!]$ THEN
         $s_2^4 := s_2^4 \cup \{r_2\}$
       END
     END
  END
END

Вышеприведенные рассуждения очевидным образом обобщаются на случай произвольного числа входных и выходных параметров функции перехода.

3.5 Построение расширений ядра канонической модели

Расширения ядра канонической модели процессов будем строить для образцов потоков работ, определенных в [4]. Каждому образцу ставится в соответствие родовой тип скрипта. Этот тип фиксирует правила, в соответствии с которыми происходит управление процессом (control flow). При этом ряд элементов скрипта (используемые функции и типы данных) являются параметрами типа.

Всюду без ограничения общности используется естественно необходимое для описания минимальное число элементов скрипта (например, два перехода, две параллельные ветви). Родовые типы скриптов имеют параметры, описываемые и связанные между собой следующим образом. Каждый переход (например, `Trunk` в образце расщепления процесса, описываемом ниже) параметризуется типом функции (в нашем случае – `trunk`), которая вызывается при срабатывании этого перехода. Каждое место (например, `entrance` в том же образце) параметризуется типом токена, допустимым для данного места (здесь – `entranceTokenType`). Функция, подставляемая в качестве параметра скрипта (например, `trunk`), должна иметь определенное количество входных и выходных параметров определенных типов (в нашем случае – один входной параметр типа `entranceTokenType` и два выходных параметра типов `auxPlace1TokenType` и `auxPlace2TokenType`).

Графическому изображению скриптов соответствует двудольный граф с двумя видами вершин – местами (изображаемыми кругами) и переходами (изображаемыми квадратами). Вершины разных видов соединяются отношением инцидентности (стрелки). В местах могут накапливаться токены различных типов. Переходы могут срабатывать при выполнении определенных условий, поглощая токены из входных мест перехода и производя токены в его выходных местах.

В настоящей работе приняты следующие условные обозначения. Входные и выходные места образца называются `entrance` и `exit` соответственно (с добавлением номера в случае необходимости). Дополнительные места носят названия `auxPlace`, `auxPlace1`, `auxPlace2` и т.д. Переходы называются либо `Trans`, либо, если образец связан с ветвлением, «стволовой» переход называется `Trunk`, а переходы в ветвях – `Branch`. В случае необходимости также вводится дополнительная нумерация. Такие нотационные соглашения не накладывают никаких ограничений на порядок использования и реализации образцов и введены для удобочитаемости.

3.5.1 Расширения, соответствующие основным образцам потока управления

Образец последовательного процесса (Sequence). Этот образец описывает ситуацию, когда несколько переходов срабатывают последовательно в определенном порядке. Каждый последующий переход может быть инициирован только после завершения предыдущего. В канонической модели данному образцу соответствует родовой тип скрипта `sequence` (рис. 3.1).

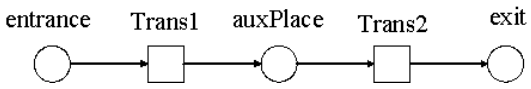


Рис. 3.1. Образец последовательного процесса

Для данного образца приводится подробная информация о параметрах и типах. Далее для краткости описание образцов ограничивается ссылками на общие рассуждения, приведенными в начале раздела, и замечаниями, необходимыми для понимания семантики канонической модели процессов.

Тип `sequence` имеет два параметра — функции `trans1` и `trans2`. Они задают функции в последовательных переходах скрипта. Функции, подставляемые в качестве параметров этого образца, должны иметь один входной параметр и один выходной параметр. Для `trans1` параметры должны иметь соответственно типы `entranceTokenType` и `auxPlaceTokenType`, для `trans2` — `auxPlaceTokenType` и `exitTokenType`.

```
{ sequence; in: script;
  params: {trans1/function,
           trans2/function,
           entranceTokenType/type,
           auxPlaceTokenType/type,
           exitTokenType/type
          };
  states:
   {entrance; token: entranceTokenType},
   {auxPlace; token: auxPlaceTokenType},
   {exit; token: exitTokenType};
  transitions:
   { Trans1;
     from: entrance;
     bind_from: {entrance, in};
     to: auxPlace;
     bind_to: { auxPlace, out};
```

```
    activity: {in: function;
     params: {+in/entranceTokenType, -out/auxPlaceTokenType};
     {{ trans1(in, out)}};
    }
  },
  { Trans2;
    from: auxPlace;
    bind_from: {auxPlace, in};
    to: exit;
    bind_to: {exit, out};
    activity: {in: function;
    params: {+in/auxPlaceTokenType, -out/exitTokenType};
    {{ trans2(in, out)}};
    }
  };
}
```

Образец расщепления процесса (AND-split). Этот образец описывает ситуацию, когда один поток управления разделяется на несколько потоков, выполняемых параллельно. В канонической модели данному образцу соответствует родовой тип скрипта `andSplit` (рис. 3.2).

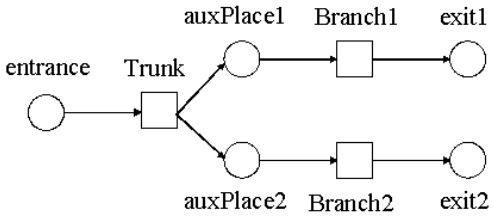


Рис. 3.2. Образец расщепления процесса

В результате срабатывания перехода `Trunk` токены будут помещены во все выходные места этого перехода (`auxPlace1` и `auxPlace2`).

```
{ andSplit; in: script;
  params: {trunk/function,
           branch1/function,
           branch2/function,
           entranceTokenType/type,
           auxPlace1TokenType/type,
           auxPlace2TokenType/type,
           exit1TokenType/type,
           exit2TokenType/type
           };
  states:
```

```
    {entrance; token: entranceTokenType},
    {auxPlace1; token: auxPlace1TokenType},
    {auxPlace2; token: auxPlace2TokenType},
    {exit1; token: exit1TokenType},
    {exit2; token: exit2TokenType};
  transitions:
    { Trunk;
        from: entrance;
        bind_from: {entrance, in};
        to: auxPlace1, auxPlace2;
        bind_to: { auxPlace1, out1}, { auxPlace2, out2};
        activity: {in: function;
        params: {+in/entranceTokenType,
                 -out1/auxPlace1TokenType,
                 -out2/auxPlace2TokenType
                };
            {{trunk(in,out1,out2)}};
        }
    },
    { Branch1;
        from: auxPlace1;
        bind_from: {auxPlace1, in};
        to: exit1;
        bind_to: {exit1, out};
        activity: {in: function;
                params: {+in/auxPlace1TokenType,
                -out/exit1TokenType
                };
            {{branch1(in, out)}};
        }
    },
    { Branch2;
        from: auxPlace2;
        bind_from: {auxPlace2, in};
        to: exit2;
        bind_to: {exit2, out};
        activity: {in: function;
        params: {+in/auxPlace2TokenType,
                -out/exit2TokenType
                };
            {{branch2(in, out)}};
        }
    }
}
```

Образец синхронизации ветвей процесса (AND-join). Этот образец описывает ситуацию, когда несколько параллельных потоков сливаются в один, ожидая при этом завершения всех входящих ветвей. В канонической модели данному образцу соответствует родовой тип скрипта `andJoin` (рис. 3.3).

Семантика перехода `Trunk` такова, что он срабатывает только в

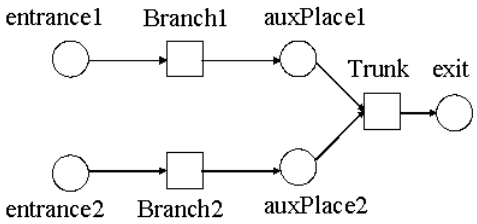


Рис. 3.3. Образец синхронизации ветвей процесса

случае, когда каждое входное место содержит токен, поэтому функция, подставляемая в качестве параметра `trunk`, должна иметь два входных параметра типов `auxPlace1TokenType` и `auxPlace2TokenType` и выходной параметр типа `exitTokenType`.

```
{ andJoin; in: script;
  params: { branch1/function,
            branch2/function,
            trunk/function,
            entrance1TokenType/type,
            entrance2TokenType/type,
            auxPlace1TokenType/type,
            auxPlace2TokenType/type,
            exitTokenType/type
  };
  states:
    {entrance1; token: entrance1TokenType},
    {entrance2; token: entrance2TokenType},
    {auxPlace1; token: auxPlace1TokenType},
    {auxPlace2; token: auxPlace2TokenType},
    {exit; token: exitTokenType};
  transitions:
    { Branch1;
        from: entrance1;
        bind_from: {entrance1, in};
        to: auxPlace1;
        bind_to: { auxPlace1, out};
        activity: {in: function;
            params: {+in/entrance1TokenType,
                     -out/auxPlace1TokenType
                    };
            {{branch1(in,out)}};
        }
    }
    { Branch2;
        from: entrance2;
        bind_from: {entrance2, in};
```

```
        to: auxPlace2;
        bind_to: { auxPlace2, out};
        activity: {in: function;
            params: {+in/entrance2TokenType,
                     -out/auxPlace2TokenType
                    };
            {{branch2(in,out)}};
        }
    }
    { Trunk;
        from: auxPlace1, auxPlace2;
        bind_from: {auxPlace1, in1}, {auxPlace2, in2};
        to: exit;
        bind_to: {exit, out};
        activity: {in: function;
            params: {+in1/auxPlace1TokenType,
                     +in2/auxPlace2TokenType,
                     -out/exitTokenType
                    };
            {{trunk(in1, in2, out)}};
        }
    }
}
```

Образец исключающего выбора (XOR-split). Этот образец описывает ситуацию, когда, в зависимости от решения, основанного на данных процесса, происходит выбор одной ветви выполнения из нескольких возможных. В канонической модели данному образцу соответствует родовой тип скрипта `xorSplit` (рис. 3.4).

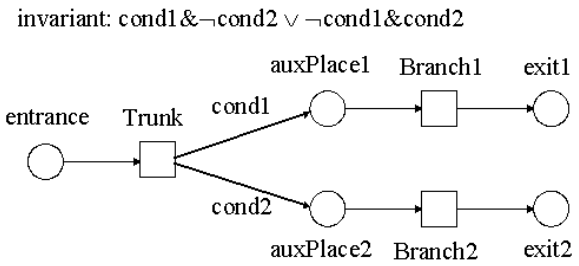


Рис. 3.4. Образец исключающего выбора

Для корректной реализации данного образца средствами скриптов, необходимо добавить в слот `instance_section` скрипта инвариант `XOR_split_invariant`, который налагает условие взаимного исключения предикатов `cond1` и `cond2`. В результате срабатывания пе-

рехода `Trunk` в соответствии с параметрами связывания, в которых на соответствующие дуги наложены условия `cond1` и `cond2`, токен будет создан в одном из выходных мест перехода `Trunk` (если все условия на дугах ложны, то переход не может быть инициирован). Таким образом, будет активизирована ровно одна из возможных ветвей.

```
{ xorSplit; in: script;
  params: { trunk/function,
            branch1/function,
            branch2/function,
            entranceTokenType/type,
            auxPlace1TokenType/type,
            auxPlace2TokenType/type,
            exit1TokenType/type,
            exit2TokenType/type
            };
  instance_section: {
    { XOR_split_invatiant;
        in: invatiant;
          {{ cond1&^cond2 | ^cond1&cond2}};
        }
    };
  states:
    {entrance; token: entranceTokenType},
    {auxPlace1; token: auxPlace1TokenType},
    {auxPlace2; token: auxPlace2TokenType},
    {exit1; token: exit1TokenType},
    {exit2; token: exit2TokenType};
  transitions:
    { Trunk;
        from: entrance;
        bind_from: {entrance, in};
        to: auxPlace1, auxPlace2;
        bind_to: { auxPlace1, out1, cond1},
                 { auxPlace2, out2, cond2};
          activity: {in: function;
            params: {+in/entranceTokenType,
                      -out1/auxPlace1TokenType,
                      -out2/auxPlace2TokenType
                     };
                {{trunk(in,out1,out2)}};
            }
    }
    { Branch1;
        from: auxPlace1;
        bind_from: {auxPlace1, in};
        to: exit1;
        bind_to: {exit1, out};
        activity: {in: function;
            params: {+in/auxPlace1TokenType,
                      -out/exit1TokenType
                     };
```

```
            {{branch1(in, out)}};
        }
    }
    { Branch2;
        from: auxPlace2;
        bind_from: {auxPlace2, in};
        to: exit2;
        bind_to: {exit2, out};
        activity: {in: function;
            params: {+in/auxPlace2TokenType,
                     -out/exit2TokenType
                    };
            {{branch2(in, out)}};
        }
    }
}
```

Образец простого слияния (XOR-join). Этот образец описывает ситуацию, когда несколько непараллельных ветвей процесса соединяются в одну. В канонической модели данному образцу соответствует родовой тип скрипта `xorJoin` (рис. 3.5).

Интерпретация образца в семантике канонической модели такова, что токен будет взят ровно из одного входного места при связывании всех входных мест с одним и тем же входным параметрам функции `trunk`. Поэтому функция, подставляемая в качестве параметра `trunk`, должна иметь один входной параметр (типа `auxPlaceTokenType`) и один выходной параметр (типа `exitTokenType`).

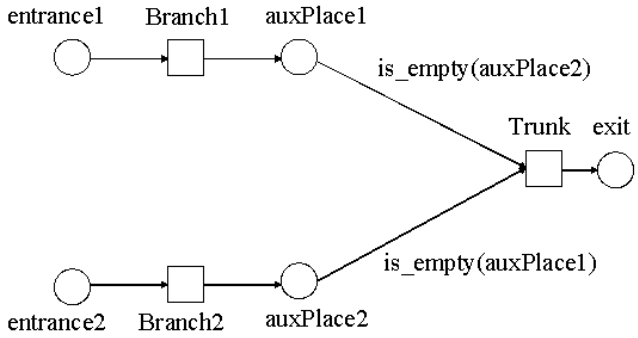


Рис. 3.5. Образец простого слияния

```
{ xorJoin; in: script;
  params: { branch1/function,
```

```
            branch2/function,
            trunk/function,
            entrance1TokenType/type,
            entrance2TokenType/type,
            auxPlaceTokenType/type,
            exitTokenType/type
          };
  states:
    {entrance1; token: entrance1TokenType},
    {entrance2; token: entrance2TokenType},
    {auxPlace1; token: auxPlaceTokenType},
    {auxPlace2; token: auxPlaceTokenType},
    {exit; token: exitTokenType};
  transitions:
    { Branch1;
        from: entrance1;
        bind_from: {entrance1, in};
        to: auxPlace1;
        bind_to: { auxPlace1, out};
        activity: {in: function;
            params: {+in/entrance1TokenType,
                     -out/auxPlaceTokenType
                    };
            {{branch1(in,out)}};
        }
    }
    { Branch2;
        from: entrance2;
        bind_from: {entrance2, in};
        to: auxPlace2;
        bind_to: { auxPlace2, out};
        activity: {in: function;
            params: {+in/entrance2TokenType,
                     -out/auxPlaceTokenType
                    };
            {{branch2(in,out)}};
        }
    }
    { Trunk;
        from: auxPlace1, auxPlace2;
        bind_from: {auxPlace1, in, is_empty(auxPlace2) },
                   {auxPlace2, in, is_empty(auxPlace1) };
        to: exit;
        bind_to: {exit, out};
        activity: {in: function;
            params: {+in/auxPlaceTokenType,
                     -out/exitTokenType
                    };
            {{trunk(in, out)}};
        }
    }
}
```

3.5.2 Расширения, соответствующие образцам сложной синхронизации и ветвления

Образец множественного выбора (OR-split). Этот образец описывает ситуацию, когда, в зависимости от решения, основанного на данных процесса, происходит выбор нескольких ветвей продолжения из множества возможных. То есть может быть выбрана как любая из этих ветвей, так и обе одновременно. В канонической модели данному образцу соответствует родовой тип скрипта `orSplit` (рис. 3.6). В результате срабатывания перехода `Trunk` токены будут созданы в выходных местах, отвечающих истинным условиям на соответствующих дугах. Таким образом, будет активизировано столько ветвей, сколько условий из `cond1` и `cond2` в параметрах связывания примет значение истина.

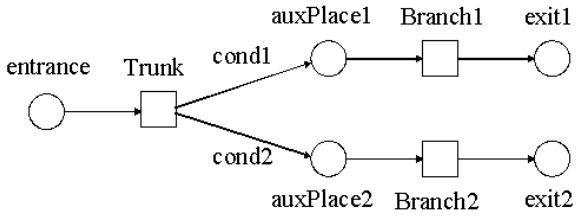


Рис. 3.6. Образец множественного выбора

```
{ orSplit; in: script;
  params: { trunk/function,
            branch1/function,
            branch2/function,
            entranceTokenType/type,
            auxPlace1TokenType/type,
            auxPlace2TokenType/type,
            exit1TokenType/type,
            exit2TokenType/type
          };
  states:
    {entrance; token: entranceTokenType},
    {auxPlace1; token: auxPlace1TokenType},
    {auxPlace2; token: auxPlace2TokenType},
    {exit1; token: exit1TokenType},
    {exit2; token: exit2TokenType};
  transitions:
    { Trunk;
        from: entrance;
```

```
        bind_from: {entrance, in};
        to: auxPlace1, auxPlace2;
        bind_to: { auxPlace1, out1, cond1},
                 { auxPlace2, out2, cond2};
        activity: {in: function;
            params: {+in/entranceTokenType,
                      -out1/auxPlace1TokenType,
                      -out2/auxPlace2TokenType
                     };
            {{trunk(in,out1,out2)}};
        }
    },
    { Branch1;
        from: auxPlace1;
        bind_from: {auxPlace1, in};
        to: exit1;
        bind_to: {exit1, out};
        activity: {in: function;
            params: {+in/auxPlace1TokenType,
                      -out/exit1TokenType
                     };
            {{branch1(in, out)}};
        }
    },
    { Branch2;
        from: auxPlace2;
        bind_from: {auxPlace2, in};
        to: exit2;
        bind_to: {exit2, out};
        activity: {in: function;
            params: {+in/auxPlace2TokenType,
                      -out/exit2TokenType
                     };
            {{branch2(in, out)}};
        }
    }
}
```

Образец синхронизированного слияния (OR-join). Этот образец описывает ситуацию, когда несколько параллельных ветвей процесса, но, возможно, не все, сливаются с синхронизацией. Графическое представление данного образца (рис. 3.7) совпадает с представлением образца *XOR-join*, за исключением отсутствия условий на дугах, входящих в переход `Trunk`. Семантика этого перехода следующая. Он может сработать при наличии токена хотя бы в одном входном месте. При срабатывании перехода поглощаются токены из максимально возможного числа входных мест.

```
{ orJoin; in: script;
```

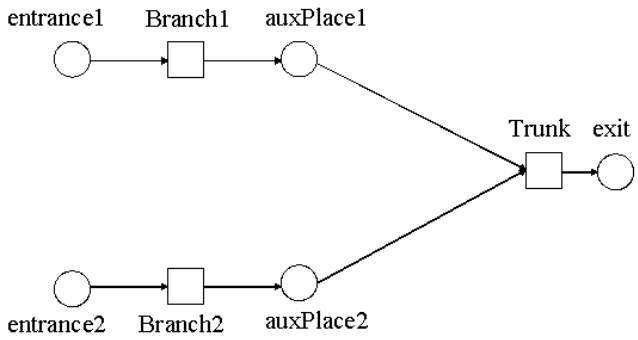


Рис. 3.7. Образец синхронизированного слияния

```
params: { branch1/function,
          branch2/function,
          trunk/function,
          entrance1TokenType/type,
          entrance2TokenType/type,
          auxPlace1TokenType/type,
          auxPlace2TokenType/type,
          exitTokenType/type
        };
states:
  {entrance1; token: entrance1TokenType},
  {entrance2; token: entrance2TokenType},
  {auxPlace1; token: auxPlace1TokenType},
  {auxPlace2; token: auxPlace2TokenType},
  {exit; token: exitTokenType};
transitions:
  { Branch1;
      from: entrance1;
      bind_from: {entrance1, in};
      to: auxPlace1;
      bind_to: { auxPlace1, out};
      activity: {in: function;
          params: {+in/entrance1TokenType,
                   -out/auxPlace1TokenType
                  };
          {{branch1(in,out)}};
      }
  },
  { Branch2;
      from: entrance2;
      bind_from: {entrance2, in};
      to: auxPlace2;
      bind_to: { auxPlace2, out};
```

```
        activity: {in: function;
            params: {+in/entrance2TokenType,
                     -out/auxPlace2TokenType
                    };
            {{branch2(in,out)}};
        }
    },
    { Trunk;
        from: auxPlace1, auxPlace2;
        bind_from:
         {auxPlace1, in1, ^isempty(auxPlace1)},
         {null, in1, isempty(auxPlace1)},
         {auxPlace2, in2, ^isempty(auxPlace2)},
         {null, in2, isempty(auxPlace2)};
        to: exit;
        bind_to: {exit, out};
        conditions: {^isempty(auxPlace1) | ^isempty(auxPlace2)};
        activity: {in: function;
        params: {+in1/auxPlace1TokenType,
                 +in2/auxPlace2TokenType,
                 -out/exitTokenType
                };
            {{trunk(inSet, out)}};
        }
    }
}
```

Образец множественного слияния (Multiple Merge). Этот образец описывает ситуацию, когда несколько, возможно, параллельных ветвей процесса сливаются без синхронизации. Графическое представление данного образца совпадает с представлением образца синхронизированного слияния (рис. 3.7).

Переход `Trunk` может сработать, поглощая один токен из одного входного места. При этом не важно, содержатся ли токены в других входных местах. Таким образом, переход `Trunk` будет срабатывать всякий раз, когда одно из входных мест содержит подходящий токен. В канонической модели данному образцу соответствует родовой тип скрипта `multiMerge`, который отличается от родового типа скрипта `xorJoin` отсутствием условий на дугах.

```
{ multiMerge; in: script;
  params: { branch1/function,
            branch2/function,
            trunk/function,
            entrance1TokenType/type,
            entrance2TokenType/type,
            auxPlaceTokenType/type,
            exitTokenType/type
```

```
          };
  states:
    {entrance1; token: entrance1TokenType},
    {entrance2; token: entrance2TokenType},
    {auxPlace1; token: auxPlaceTokenType},
    {auxPlace2; token: auxPlaceTokenType},
    {exit; token: exitTokenType};
  transitions:
    { Branch1;
        from: entrance1;
        bind_from: {entrance1, in};
        to: auxPlace1;
        bind_to: { auxPlace1, out};
        activity: {in: function;
            params: {+in/entrance1TokenType,
                     -out/auxPlaceTokenType
                    };
            {{branch1(in,out)}};
        }
    },
    { Branch2;
        from: entrance2;
        bind_from: {entrance2, in};
        to: auxPlace2;
        bind_to: { auxPlace2, out};
        activity: {in: function;
            params: {+in/entrance2TokenType,
                     -out/auxPlaceTokenType
                    };
            {{branch2(in,out)}};
        }
    },
    { Trunk;
        from: auxPlace1, auxPlace2;
        bind_from: {auxPlace1, in},
                   {auxPlace2, in};
        to: exit;
        bind_to: {exit, out};
        activity: {in: function;
            params: {+in/auxPlaceTokenType,
                     -out/exitTokenType
                    };
            {{trunk(in, out)}};
        }
    }
}
```

Образец дискриминатора (Discriminator). Этот образец описывает ситуацию, когда ожидается завершение одной из нескольких параллельных ветвей процесса. В результате этого активизируется по-

следующий переход, а все остальные параллельные ветви игнорируются. Следуя [5], полагаем, что при завершении одной ветви остальные не просто игнорируются, а отменяются. В канонической модели данному образцу соответствует родовой тип скрипта `discriminator` (рис. 3.8). На рисунке с переходом `Trunk` пунктирной линией соединен пунктирный прямоугольник со скругленными углами. Это означает, что при срабатывании перехода `Trunk` из отмеченной таким образом области удаляются все токены. Таким образом реализуется отмена (подробнее об образце отмены см. раздел 3.5.3).

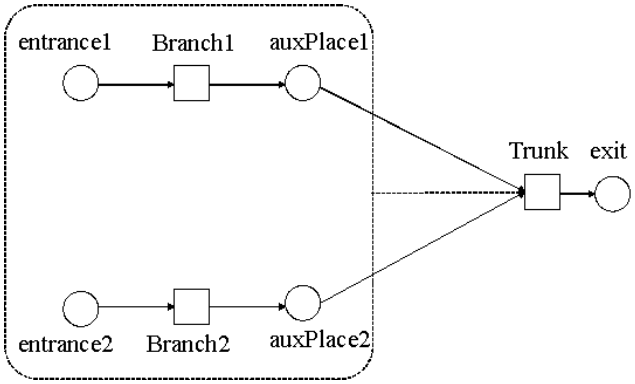


Рис. 3.8. Образец дискриминатора

Родовой тип скрипта, отвечающий данному образцу в канонической модели, можно представить как результат композиции родовых типов скриптов для образцов множественного слияния и прекращения деятельности.

Тип имеет три параметра-функции. Параметры `branch1` и `branch2` соответствуют функциям параллельных ветвей, а параметр `trunk` соответствует функции перехода, который срабатывает по завершении одной из ветвей, при этом отменяя выполнение другой. Реализация образца осуществляется с использованием конструкции отмены, подробно разбираемой в образцах отмены. Типы входных и выходных параметров функций, подставляемых в качестве параметров `branch1`, `branch2` и `trunk`, должны быть такими же, как для образца `multiMerge`.

```
{ discriminator; in: script;
  params: { branch1/function,
            branch2/function,
```

```
          trunk/function,
          entrance1TokenType/type,
          entrance2TokenType/type,
          auxPlaceTokenType/type,
          exitTokenType/type
         };
states:
  {entrance1; token: entrance1TokenType},
  {entrance2; token: entrance2TokenType},
  {auxPlace1; token: auxPlaceTokenType},
  {auxPlace2; token: auxPlaceTokenType},
  {exit; token: exitTokenType};
transitions:
  { Branch1;
      from: entrance1;
      bind_from: {entrance1, in};
      to: auxPlace1;
      bind_to: { auxPlace1, out};
      activity: {in: function;
          params: {+in/entrance1TokenType,
                   -out/auxPlaceTokenType
                  };
          {{branch1(in,out)}};
      }
  },
  { Branch2;
      from: entrance2;
      bind_from: {entrance2, in};
      to: auxPlace2;
      bind_to: { auxPlace2, out};
      activity: {in: function;
          params: {+in/entrance2TokenType,
                   -out/auxPlaceTokenType
                  };
          {{branch2(in,out)}};
      }
  },
  { Trunk;
      from: auxPlace1, auxPlace2;
      bind_from: {auxPlace1, in}, {auxPlace2, in};
      to: exit;
      bind_to: {exit, out};
      activity: {in: function;
          params: {+in/auxPlaceTokenType,
                   -out/exitTokenType
                  };
          {{
             trunk(in, out) &
             isempty(entrance1') & isempty(entrance2') &
             isempty(auxPlace1') & isempty(auxPlace2')
          }};
      }
  }
```

```
}
```

3.5.3 Расширения, соответствующие образцам прекращения деятельностей

Эта группа расширений включает в себя образцы *Cancel Activity* и *Cancel Case*. Первый описывает ситуацию, когда отменяется готовый к выполнению или выполняемый переход. Второй образец описывает ситуацию, когда отменяется экземпляр процесса. Данные образцы имеют сходную природу, в результате чего могут быть реализованы с помощью некоторого унифицированного построения, которое графически изображается следующим образом. Отменяемая часть процесса изображается пунктирным прямоугольником со скругленными углами, который соединяется с отменяющим эту часть процесса переходом пунктирной линией.

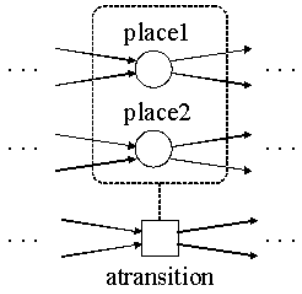


Рис. 3.9. Прекращение деятельностей

В канонической модели данной группе расширений соответствует родовой тип скрипта `cancel` (рис. 3.9). Тип имеет один параметр-скрипт `ascript`, параметр-переход `atransition` и два параметра-места `place1` и `place2`. Отмена реализуется следующим образом. Для этого образца отмена задается дополнительными действиями перехода скрипта, в соответствии с которыми в данном скрипте при срабатывании этого перехода удаляются токены из указанной пары мест. Синтаксически этому соответствует добавление в тело функции, связанной с переходом, предикатов `isempty(place1')` и `isempty(place2')`. Данные предикаты означают следующее. В соответствии с семантикой канонической модели, действие функции на пространстве состояний системы задается при помощи формулы, связывающей состояние

системы перед выполнением функции и после выполнения функции, т.е. формула представляет собой смешанные пред- и постусловия. При этом сущности, отвечающие постсостоянию системы, помечаются в формуле апострофом. Места скриптов в канонической модели рассматриваются как множества токенов. Предикат `isempty(place1')` означает, что постусловием для функции, связанной с переходом `atransition`, является совпадение множества `place1` с пустым множеством.

```
{ cancel; in: script;
  params: { ascript/script, atransition/transition, place1, place2 };
  supertype: ascript;
  transitions:
    { atransition;
        from: atransition.from;
        bind_from: atransition.bind_from;
        to: atransition.to;
        bind_to: atransition.bind_to;
        activity:{ in: function;
            params: atransition.activity.params;
            {{
               atransition.activity(atransition.activity.params) &
               isempty(place1') & isempty(place2')
            }}
        }
    };
}
```

3.5.4 Расширения, соответствующие образцам, использующим концепцию состояния

Образец отложенного выбора (Deferred Choice). Этот образец описывает ситуацию, когда происходит выбор одной ветви процесса из нескольких возможных. При этом выбор осуществляет среда процесса. В канонической модели данному образцу соответствует родовой тип скрипта `deferredChoice` (рис. 3.10).

```
{ deferredChoice; in: script;
  params: { trunk/function,
            branch1/function,
            branch2/function,
            entranceTokenType/type,
            cnoicePlaceTokenType/type,
            exit1TokenType/type,
            exit2TokenType/type
          };
  states:
```

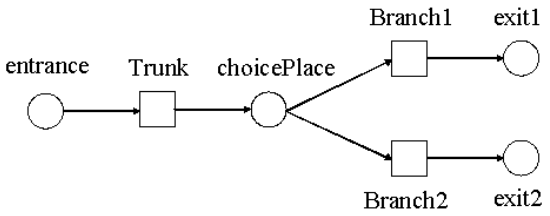


Рис. 3.10. Образец отложенного выбора

```
  {entrance; token: entranceTokenType},
  {cnoicePlace; token: cnoicePlaceTokenType},
  {exit1; token: exit1TokenType},
  {exit2; token: exit2TokenType};
transitions:
  { Trunk;
      from: entrance;
      bind_from: {entrance, in};
      to: cnoicePlace;
      bind_to: { cnoicePlace, out};
      activity: {in: function;
          params: {+in/entranceTokenType,
                   -out/cnoicePlaceTokenType
                  };
          {{trunk(in,out)}};
      }
  },
  { Branch1;
      from: cnoicePlace;
      bind_from: { cnoicePlace, in};
      to: exit1;
      bind_to: {exit1, out};
      activity: {in: function;
      params: {+in/ cnoicePlaceTokenType,
               -out/exit1TokenType
              };
          {{branch1(in, out)}};
      }
  },
  { Branch2;
      from: cnoicePlace;
      bind_from: { cnoicePlace, in};
      to: exit2;
      bind_to: {exit2, out};
      activity: {in: function;
      params: {+in/ cnoicePlaceTokenType,
               -out/exit2TokenType
```

```
            };
        {{branch2(in, out)}};
    }
  }
}
```

Образец поочередной параллельной маршрутизации (Interleaved Parallel Routing). Этот образец описывает ситуацию, когда несколько переходов срабатывают последовательно, но в произвольном порядке, который определяется во время исполнения. В канонической модели данному образцу соответствует родовой тип скрипта `interleavedRouting` (рис. 3.11).

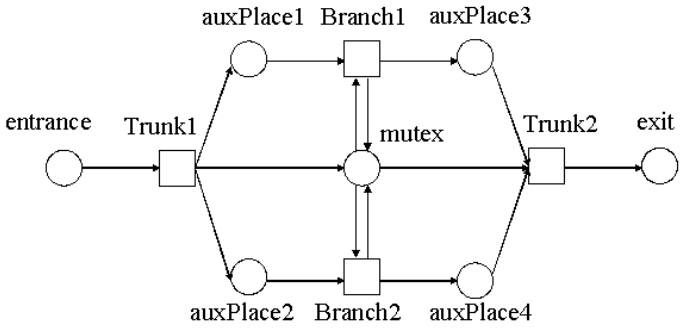


Рис. 3.11. Образец поочередного параллельного исполнения

Реализация образца осуществляется введением места `mutex`, с помощью которого переходы `Branch1` и `Branch2` исполняются последовательно, но в произвольном порядке.

```
{ interleave; in: script;
  params: { trunk1/function,
            branch1/function,
            branch2/function,
            trunk2/function,
            entranceTokenType/type,
            auxPlace1TokenType/type,
            auxPlace2TokenType/type,
            mutexTokenType/type,
            auxPlace3TokenType/type,
            auxPlace4TokenType/type,
            exitTokenType/type
```

```
        };
states:
  {entrance; token: entranceTokenType},
  {auxPlace1; token: auxPlace1TokenType},
  {auxPlace2; token: auxPlace2TokenType},
  {mutex; token: mutexTokenType},
  {auxPlace3; token: auxPlace3TokenType},
  {auxPlace4; token: auxPlace4TokenType},
  {exit; token: exitTokenType};
transitions:
  { Trunk1;
      from: entrance;
      bind_from: {entrance, in};
      to: auxPlace1, auxPlace2, mutex;
      bind_to: { auxPlace1, out1},
               { auxPlace2, out2},
               { mutex, out3};
      activity: {in: function;
          params: {+in/entranceTokenType,
                   -out1/auxPlace1TokenType,
                   -out2/auxPlace2TokenType,
                   -out3/mutexTokenType
                  };
          {{trunk1(in,out1,out2,out3)}};
      }
  },
  { Branch1;
      from: auxPlace1, mutex;
      bind_from: {auxPlace1, in1}, {mutex, in2};
      to: auxPlace3, mutex;
      bind_to: {auxPlace3, out1}, {mutex, out2};
      activity: {in: function;
          params: {+in1/auxPlace1TokenType,
                   +in2/mutexTokenType,
                   -out1/auxPlace3TokenType,
                   -out2/mutexTokenType
                  };
          {{branch1(in1, in2, out1, out2)}};
      }
  },
  { Branch2;
      from: auxPlace2, mutex;
      bind_from: {auxPlace2, in1}, {mutex, in2};
      to: auxPlace4, mutex;
      bind_to: {auxPlace4, out1}, {mutex, out2};
      activity: {in: function;
      params: {+in1/auxPlace1TokenType,
               +in2/mutexTokenType,
               -out1/auxPlace3TokenType,
               -out2/mutexTokenType
              };
          {{branch2(in1, in2, out1, out2)}};
      }
```

```
    },
    { Trunk2;
        from: auxPlace3, auxPlace4, mutex;
        bind_from: { auxPlace3, in1},
                   { auxPlace4, in2},
                   { mutex, out3};
        to: exit;
        bind_to: {exit, out};
        activity: {in: function;
            params: {+in1/auxPlace3TokenType,
                     +in2/auxPlace4TokenType,
                     +in3/mutexTokenType,
                     -out/exitTokenType
                    };
            {{trunk1(in1, in2, in3, out)}};
        }
    }
}
```

Образец вехи (Milestone). Этот образец описывает ситуацию, когда некоторый переход может сработать, только если процесс находится в определенном состоянии, т.е. если достигнута некоторая контрольная точка (веха).

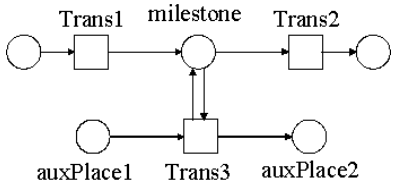


Рис. 3.12. Образец вехи

В канонической модели данный образец (рис. 3.12) реализуется путем объявления соответствующего места сети (`milestone`) как *шлюза* (`gate`), что делает его доступным извне.

```
gates:
  { milestone; token: milestoneTokenType}
```

При этом переход из другой сети (`Trans3`) может проверить наличие токенов в данном месте. Этот переход в таком случае имеет

следующую спецификацию.

```
{ Trans3;
    from: auxPlace1, milestone;
    bind_from: {auxPlace1, in1}, {milestone, in2};
    to: auxPlace2, milestone;
    bind_to: {auxPlace2, out1}, {milestone, out2};
    activity: {in: function;
        params: {+in1/auxPlace1TokenType,
                 +in2/milestoneTokenType,
                 -out1/auxPlace2TokenType,
                 -out2/milestoneTokenType
                };
        {{task3(in1, in2, out1, out2)}};
    }
}
```

3.5.5 Расширения, соответствующие образцам структурирования

Образец произвольных циклов (Arbitrary Cycles). Этот образец описывает участок в потоке работ, где один или несколько переходов могут срабатывать повторно. Исследования [5] позволяют утверждать, что каноническая модель процессов, использующая сети Петри с логическими условиями на дугах, позволяет выражать любые по сложности циклы (имеющие произвольное число точек входа и выхода, а также совмещенные с другими образцами).

Образец неявного завершения (Implicit Termination). Этот образец описывает ситуацию, когда некоторый (под-) процесс должен быть завершен, если ничего другого сделать нельзя. Другими словами, в процессе нет исполняемых переходов, и никакой переход не может сработать (и в то же время процесс не находится в состоянии тупика).

В канонической модели процессов данный образец реализуется автоматически, если разработчик потока работ корректно составляет спецификации. Корректная спецификация предполагает, что в случае ситуации, описываемой образцом неявного завершения, в выходных местах находится достаточное количество токенов, и процесс может быть завершен.

4 Модели образцов потоков работ на основе языка YAWL

YAWL (Yet Another Workflow Language) — язык описания потоков работ, разрабатываемый группой ученых во главе с Ван дер Аальстом [5] с целью определения различных образцов потоков работ. Он предоставляет конструкции, реализующие все образцы потоков работ [5]. Согласно терминологии языка YAWL, аналог места сети Петри называется *условием (condition)*. Для единообразия изложения, мы будем придерживаться терминологии, принятой для сетей Петри.

Спецификацией потока работ в языке YAWL является множество Расширенных сетей потоков работ (Extended Workflow Nets — EWF-сетей) [5], образующих древовидную иерархическую структуру. В настоящей работе используется подмножество языка YAWL, полученное элиминацией из полного языка средств реализации образцов, относящихся к множественности экземпляров и средств построения иерархических спецификаций. Расширение канонической модели для образцов, использующих средства, относящиеся к множественности экземпляров, а также средства построения иерархических спецификаций технически усложняет рассуждения, однако не добавляет методологической сложности. Поэтому, в исходных процессных моделях (образцах потоков работ) спецификацией потока работ является единственная EWF-сеть.

4.1 Синтаксис EWF сетей

Определение 4.1 Расширенная сеть потока работ [5] — *это набор*

$$N = (C, i, o, T, F, split, join, rem)$$

где

- C — *множество мест,*
- $i \in C$ — *входное место,*
- $o \in C$ — *выходное место,*
- T — *множество задач,*
- $F \subseteq (C \setminus \{o\} \times T) \cup (T \times C \setminus \{i\})$ — *отношение инцидентности,*
- *каждая вершина в графе* $(C \cup T, F)$ *содержится в направленном пути от* i *к* o*,*

- $split : T \to \{AND, XOR, OR\}$ — *определяет split-поведение каждой задачи,*
- $join : T \to \{AND, XOR\}$ — *определяет join-поведение каждой задачи,*
- $rem : T \nrightarrow P(T \cup C \setminus \{i, o\})$ — *определяет дополнительные токены, которые необходимо убрать из части потока работ,*
- $nofi : T \nrightarrow \mathbb{N} \times \mathbb{N}^{inf} \times \mathbb{N}^{inf} \times \{dynamic, static\}$ — *определяет возможность множественности экземпляров каждой задачи*

В языке YAWL под произвольной задачей $t \in T$ имеется в виду сеть Петри определенного вида. В нашем случае может быть только один тип подсети, а именно изображенный на рис. 4.1. Произвольная задача t на данной диаграмме представлена прямоугольником со скругленными углами. Фактически, она заменяется на переход $enter_t$, место $exec_t$ и переход $exit_t$. Описание функционирования данной подсети и ее связь с задачей t даны ниже. Таким образом, мы будем говорить о задаче t или о сети с элементами $\{enter_t, exec_t, exit_t\}$, имея в виду одну и ту же сущность с разной степенью детализации.

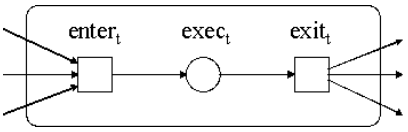


Рис. 4.1. Структура перехода EWF-сети

Оригинальная спецификация EWF-сети языка YAWL позволяет отношению инцидентности соединять две задачи непосредственно (таким образом, графически такая спецификация не соответствует двудольному графу). Так как семантика все равно подразумевает наличие неявного промежуточного места, мы, для единообразия изложения, исключим эту возможность. Также, поскольку в настоящей работе не рассматриваются средства, относящиеся к множественности экземпляров, будем считать, что $nofi = \varnothing$ для любой рассматриваемой EWF-сети.

Определение EWF-сети имеет следующие синтаксические отличия от классических сетей Петри:

- Введены два специальных места i и o — начальное и конечное.

- Функции *split*, *join*, *rem* определяют свойства каждой задачи. Первые две показывают, является ли задача AND/OR/XOR-split или AND/OR/XOR-join. *rem* — частичная функция, показывающая, какие части сети должны быть освобождены от токенов. Она может применяться не только к местам, но и к задачам. Последнее означает отмену (исполняемой) задачи.

4.2 Семантика EWF сетей

Пространство состояний. Пространство состояний спецификации потока работ определяется на основании аналогии с понятием *токен* в раскрашенных сетях Петри. Пространство состояний состоит из набора токенов, содержащих данные. Поскольку следуя [5], здесь данные не рассматриваются, то достаточно считать, что для токенов есть понятие их *идентифицируемости*. Для работы с одинаково идентифицируемыми токенами в одном и том же месте используется понятие мультимножества. Множество мультимножеств над множеством s обозначается $B(s)$.

Определение 4.2 Состояние потока работ *спецификации*

$$N = (C, i, o, T, F, split, join, rem)$$

это мультимножество s над Q, где $Q = C \cup (\cup_{t \in T} exec_t)$, т.е. $s \in B(Q)$. Далее состояние потока работ будем называть просто состоянием.

Состояние s потока работ — это мультимножество токенов, где для каждого токена $(x) \in s$, x обозначает местоположение (location) токена. Оно является либо (1) местом ($x \in Q$), либо (2) местом в подсети, отвечающей некоторой задаче $t \in T$ (этому понятию соответствует состояние задачи (task state) в [5]). В нашем случае каждая задача может иметь только одно место в отвечающей ей подсети (рис. 4.1), соответствующее нахождению задачи t в стадии исполнения (executing), при этом $x = exec_t$.

Переходы на пространстве состояний. Для задачи t все переходы на пространстве состояний можно описать диаграммой, изображенной на рис. 4.1.

Существует два типа перехода в подсети, отвечающей произвольной задаче t: $enter_t$ и $exit_t$. Также в ней существует место $exec_t$.

Когда наступает время выполнить задачу t, срабатывает соответствующий ей переход $enter_t$. При этом необходимо выполнение опре-

деленных условий на наличие токенов во входных местах, в зависимости от того, является ли задача t XOR/OR/AND-join. В результате срабатывания перехода $enter_t$ создается токен в месте $exec_t$. Нахождение токена в месте $exec_t$ означает, что задача t находится в стадии исполнения.

Переход $exit_t$ срабатывает, когда токен находится в месте $exec_t$. При этом переход удаляет токены из выбранных частей сети в соответствии с функцией rem. Число производимых токенов зависит от split-поведения задачи t.

Определение 4.3 *Пусть* $N = (C, i, o, T, F, split, join, rem, nofi)$ — *спецификация EWF-сети. Функции* $\bullet\_, \_\bullet : (C \cup T) \to P(C \cup T)$, *определяют для каждой вершины множества вершин, непосредственно связанных с данной отношением инцидентности. Для вершины* $x \in C \cup T$, $\bullet x = \{y \mid (y, x) \in F\}$ — *множество входящих вершин и* $x\bullet = \{y \mid (x, y) \in F\}$ — *множество выходящих вершин.*

Следующие определения формализуют множество переходов, возможных в данном состоянии, при помощи *отношений связывания* (binding relations; далее — связываний для краткости). Первое из них — это $binding_{enter}$. $binding_{enter}(t, c, p, s)$ является логическим выражением, которое принимает значение истина, если переход $enter$ может сработать для задачи t, в состоянии s, с поглощением мультимножества токенов c и созданием мультимножества токенов p.

Определение 4.4 *Пусть* $N = (C, i, o, T, F, split, join, rem)$ — *спецификация EWF-сети и* $t \in T$, c, p, $s \in B(Q)$. *Логическая функция* $binding_{enter}(t, c, p, s)$ *принимает значение истина, если и только если выполняются следующие условия:*

- *токены, которые должны быть поглощены, присутствуют в входных местах задачи* t*:*

$$c \subseteq s$$

- *токены поглощаются из входных мест задачи* t, *и не более одного токена может быть поглощено из каждого входного места:*

$$c \subseteq \bullet t$$

- *токены, которые должны быть произведены, не присутствуют в выходных местах задачи* t, *т.е. исполняемая задача не может быть выполнена повторно до своего завершения:*

$$s \cap p = \varnothing$$

- *для поведения AND-join, все входные места содержат токены:*

$$join(t) = AND \Rightarrow q(c) = \bullet t$$

- *для поведения XOR-join, только одно входное место содержит токены, кроме того, это входное место содержит не более одного токена:*

$$join(t) = XOR \Rightarrow q(c) — \text{синглетон}$$

Определение 4.5 *Пусть* $N = (C, i, o, T, F, split, join, rem)$ *— спецификация EWF-сети и* $t \in T$, c, p, $s \in B(Q)$. *Логическая функция* $binding_{exit}(t, c, p, s)$ *принимает значение истина, если и только если выполняются следующие условия:*

- *токены, которые должны быть поглощены, присутствуют в входных местах задачи* t*:*

$$c \subseteq s$$

- *токены производятся только в выходных местах рассматриваемой задачи* t*, и не более одного токена может быть произведено в каждом выходном месте:*

$$p \subseteq t\bullet$$

- *для поведения AND-split, токены производятся во всех выходных местах:*

$$split(t) = AND \Rightarrow q(p) = t\bullet$$

- *для поведения OR-split, токены производятся в некоторых выходных местах:*

$$split(t) = OR \Rightarrow q(p) \neq \varnothing$$

- *для поведения XOR-split, единственный токен производится ровно в одном выходном месте рассматриваемой задачи:*

$$split(t) = XOR \Rightarrow q(p) \text{ — синглетон}$$

- *поглощаются все токены из участка сети, указанного в соответствующем отношении отмены:*

$$s - c \cap rem(t) = \varnothing$$

Определение 4.6 *Пусть* $N = (C, i, o, T, F, split, join, rem)$ *— спецификация EWF-сети и* $t \in T$, c, p, $s \in B(Q)$. *Логическая функция* $binding(t, c, p, s)$ *принимает значение истина, если и только если выполняются одно из следующих условий:*

- *разрешено исполнение перехода enter в подсети, соответствующей задаче* t*:*
$$binding_{enter}(t, c, p, s)$$
- *разрешено исполнение перехода exit в подсети, соответствующей задаче* t*:*
$$binding_{exit}(t, c, p, s).$$

Отношение перехода на пространстве состояний потока работ определяется с использованием $binding(t, c, p, s)$.

Определение 4.7 *Пусть* $N = (C, i, o, T, F, split, join, rem)$ *— спецификация EWF-сети, а* s_1 *и* s_2 *— два состояния EWF-сети* N. *Тогда* $s_1 \rightarrow s_2$ *если, и только если существуют* $t \in T, c, p \in B(Q)$, *такие, что* $binding(t, c, p, s_1)$ *и* $s_2 = (s_1 - c) \cup p$.

4.3 Отображение расширенных сетей потоков работ в AMN

Рассмотрим расширенную сеть потока работ

$$N = \langle C, i, o, T, F, split, join, rem \rangle$$

Такая сеть представляется в AMN конструкцией *REFINEMENT* с именем N. Это связано с тем, что в процессе расширения канонической модели спецификации YAWL всегда выступают в роли уточняющих, а значит не могут быть представлены в AMN конструкцией *MACHINE*. Кроме того, абстрактный уровень спецификаций YAWL требует использования недетерминированных подстановок, что исключает возможность использования конструкции *IMPLEMENTATION* для представления спецификаций YAWL в AMN.

Множество мест $C = \{c_1, \ldots, c_n\}$, вместе с входным и выходным местом сети, представляется в AMN[1] переменной *states* машины N. Переменная типизируется в разделе *INVARIANT* машины N следующим образом.

[1]Исключая средства представления множественности экземпляров в EWF-сети.

$states \in States \rightarrow NAT$

Здесь множество $States$, определяемое в разделе $SETS$ машины N как

$States = \{state\_input, state\_output, state\_c_1, \ldots, state\_c_n\}$

представляет собой множество строковых констант, соответствующих именам мест.

Натуральное число, хранимое в $states(state\_c_i)$, отражает количество токенов, содержащихся в месте c_i сети. Переменная $states$ инициализируется в разделе $INITIALISATION$ машины N следующим образом.

ANY $states1$
WHERE
$states1 \in States \rightarrow NAT \wedge$
$\forall st \bullet (st \in \mathrm{dom}(states1) \Rightarrow states1(st) = 0)$
THEN
$states := states1$
END

Каждая задача $t_i \in T$ представляется в AMN двумя операциями $enter\_t_i$ и $exit\_t_i$, а также переменной $exec\_t_i$ машины N. Переменная $exec\_t_i$ типизируется в разделе $INVARIANT$ машины N следующим образом.

$exec\_t_i \in NAT$

Пусть множество входных мест задачи t_i,

$$\bullet\, t_i = \{c \mid \langle c, t_i \rangle \in F\} = \{c_1^{in}, \ldots, c_m^{in}\}$$

Пусть множество выходных мест задачи t_i,

$$t_i \bullet = \{c \mid \langle c, t_i \rangle \in F\} = \{c_1^{out}, \ldots, c_k^{out}\}$$

Пусть множество мест, из которых нужно убрать токены по исполнении задачи t_i

$$rem(t_i) = \{c_1^{rem}, \ldots, c_l^{rem}\} \cup \{t_1^{rem}, \ldots, t_r^{rem}\}$$

где $c_1^{rem}, \ldots, c_l^{rem}$ из C, $t_1^{rem}, \ldots, t_r^{rem}$ из T.

Операция $enter\_t_i$ машины N имеет следующий вид.

$enter\_t_i =$
SELECT $exec\_t_i = 0 \wedge P_{enter}$
THEN
$S_{enter} \;||$
$exec\_t_i = exec\_t_i + 1$
END

Вид предиката P_{enter} зависит от join-поведения задачи t_i (значения $join(t_i)$).

- $join(t_i) = AND \Rightarrow$
  $P_{enter} \leftrightharpoons states(state\_c_1^{in}) > 0 \wedge \ldots \wedge states(state\_c_m^{in}) > 0$

- $join(t_i) = OR \Rightarrow$
  $P_{enter} \leftrightharpoons states(state\_c_1^{in}) > 0 \vee \ldots \vee states(state\_c_m^{in}) > 0$

- $join(t_i) = XOR \Rightarrow$
  $P_{enter} \leftrightharpoons$
  $states(state\_c_1^{in}) > 0 \wedge$
  $states(state\_c_2^{in}) = 0 \wedge \ldots \wedge states(state\_c_m^{in}) = 0$
  $\vee \ldots \vee$
  $states(state\_c_1^{in}) = 0 \wedge \ldots \wedge states(state\_c_{j-1}^{in}) = 0 \wedge$
  $states(state\_c_j^{in}) > 0 \wedge$
  $states(state\_c_{j+1}^{in}) = 0 \wedge \ldots \wedge states(state\_c_m^{in}) = 0$
  $\vee \ldots \vee$
  $states(state\_c_1^{in}) = 0 \wedge \ldots \wedge states(state\_c_{m-1}^{in}) = 0 \wedge$
  $states(state\_c_m^{in}) > 0$

Вид подстановки S_{enter} также зависит от join-поведения задачи.

- $join(t_i) = AND \Rightarrow$
  $S_{enter} \leftrightharpoons$
  $states := states \Leftarrow$
  $\{state\_c_1^{in} \mapsto states(state\_c_1^{in}) - 1,$
  $\ldots,$
  $state\_c_m^{in} \mapsto states(state\_c_m^{in}) - 1\}$

- $join(t_i) = XOR \Rightarrow$
  $S_{enter} \leftrightharpoons$

IF $states(state\_c_1^{in}) > 0$
THEN $states(state\_c_1^{in}) := states(state\_c_1^{in}) - 1$
ELSIF $states(state\_c_m^{in}) > 0$
THEN $states(state\_c_m^{in}) := states(state\_c_m^{in}) - 1$
END

- $join(t_i) = OR \Rightarrow$
  $S_{enter} \leftrightharpoons$
  $states := states \lhd\!\!+$
  $\{st, num \mid st \in \{state\_c_1^{in}, \ldots, state\_c_m^{in}\} \wedge$
  $states(st) > 0 \wedge num = states(st) - 1\}$

Операция $exit\_t_i$ машины N имеет следующий вид.

$exit\_t_i =$
SELECT $exec\_t_i > 0$
THEN
$(S_{exit} \mid\mid exec\_t_i = exec\_t_i - 1);\ R_{exit}$
END

Вид подстановки S_{exit} зависит от split-поведения задачи t_i (значения $split(t_i)$).

- $join(t_i) = AND \Rightarrow$
  $S_{exit} \leftrightharpoons$
  $states := states \lhd\!\!+$
  $\{state\_c_1^{out} \mapsto states(state\_c_1^{out}) - 1,$
  $\ldots,$
  $state\_c_k^{out} \mapsto states(state\_c_k^{out}) - 1\}$

- $join(t_i) = XOR \Rightarrow$
  $S_{exit} \leftrightharpoons$
  ANY st WHERE $st \in States$
  THEN
  $states(st) := states(st) + 1$
  END

- $join(t_i) = OR \Rightarrow$
  $S_{exit} \leftrightharpoons$
  ANY sts WHERE $sts \in \mathbb{P}(States) \wedge \mathrm{card}\,(sts) > 0$

THEN
$states := states \lhd \{st, num \mid st \in sts \wedge num = states(st) + 1\}$
END

Подстановка R_{exit} имеет следующий вид.

$R_{exit} \leftleftarrows$
$states :=$
$states \lhd \{st, num \mid st \in \{states\_c_1^{rem}, \ldots, states\_c_l^{rem}\} \wedge num = 0\} \parallel$
$exec\_t_1^{rem} := 0 \parallel \ldots \parallel exec\_t_r^{rem} := 0$

4.4 Корректность уточняющего расширения канонической модели процессов

Рассмотрим процесс корректного уточняющего расширения канонической модели процессов на примере образца потоков работ – дискриминатора.

Компоненты дискриминатора, как EWF-сети, выглядят следующим образом.

$Discriminator = \langle C, i, o, T, F, split, join, rem \rangle$

$C = \{enter1, enter2, auxState1, auxState2, exit\}$
$T = \{branch1, branch2, trunk\}$
$F = \{enter1 \mapsto branch1, enter2 \mapsto branch2,$
$branch1 \mapsto auxSatate1, branch2 \mapsto auxSatate2,$
$auxSatate1 \mapsto trunck, auxSatate2 \mapsto trunck,$
$trunk \mapsto exit\}$
$split = \varnothing$
$join = \{trunk \mapsto XOR\}$
$rem = \{trunck \mapsto$
$\{enter1, enter2, auxState1, auxState2, branch1, branch2\}\}$

Данная EWF-сеть представляется в AMN следующей конструкцией.

REFINEMENT *Discriminator*

SETS
States =

$\{state\_enter1, state\_enter2,$
$state\_auxState1, state\_auxState2, state\_exit\}$

VARIABLES
$states,$
$exec\_branch1, exec\_branch2, exec\_trunk$
INVARIANT
$states \in States \to NAT \wedge$
$exec\_branch1 \in NAT \wedge exec\_branch2 \in NAT \wedge exec\_trunk \in NAT$
INITIALISATION
ANY $states1$
WHERE
$states1 \in States \to NAT \wedge$
$\forall st \bullet (st \in \mathrm{dom}(states1) \Rightarrow states1(st) = 0)$
THEN
$states := states1$
END
||
$exec\_branch1 := 0 \,||\, exec\_branch2 := 0 \,||\, exec\_trunk := 0$

OPERATIONS

$enter\_branch1 =$
SELECT $exec\_branch1 = 0 \wedge states(state\_enter1) > 0$
THEN
$states(state\_enter1) := states(state\_enter1) - 1 \,||$
$exec\_branch1 := exec\_branch1 + 1$
END ;

$exit\_branch1 =$
SELECT $exec\_branch1 > 0$
THEN
$states(state\_auxState1) := states(state\_auxState1) + 1 \,||$
$exec\_branch1 := exec\_branch1 - 1$
END

$enter\_branch2 =$
SELECT $exec\_branch2 = 0 \wedge states(state\_enter2) > 0$
THEN
$states(state\_enter2) := states(state\_enter2) - 1 \,||$
$exec\_branch2 := exec\_branch2 + 1$
END

$exit\_branch2 =$
SELECT $exec\_branch2 > 0$
THEN
$states(state\_auxState2) := states(state\_auxState2) + 1 \;||$
$exec\_branch2 := exec\_branch2 - 1$
END

$enter\_trunk =$
SELECT
$exec\_trunk = 0 \wedge$
$(states(state\_auxState1) > 0 \wedge states(state\_auxState2) = 0 \vee$
$(states(state\_auxState1) = 0 \wedge states(state\_auxState2) > 0)$
THEN
IF $states(state\_auxState1) > 0$
THEN
$states(state\_auxState1) := states(state\_auxState1) - 1$
ELSIF $states(state\_auxState2) > 0$
THEN
$states(state\_auxState2) := states(state\_auxState2) - 1$
END
$||$
$exec\_trunk := exec\_trunk + 1$
END

$exit\_trunk =$
SELECT $exec\_trunk > 0$
THEN
$(state\_exit := state\_exit + 1 \;||$
$exec\_trunk := exec\_trunk - 1);$
$(states :=$
$states \lhd \{st \mapsto num \mid num = 0 \wedge$
$st \in \{state\_enter1, state\_enter2,$
$state\_auxState1, state\_auxState2\}\} \;||$
$exec\_branch1 := 0 \;||$
$exec\_branch2 := 0)$
END

END

Расширение канонической модели процессов для реализации образца *Discriminator* осуществляется при помощи родового типа скрип-

та, являющегося частичной конкретизацией родового типа скрипта `cancel` следующего вида.

```
{ cancel;
  params: {
   xorJoin,
   trunk,
   entrance1,
   entrance2,
   auxState1,
   auxState2
  };
}
```

Этот родовой тип скрипта представляется в AMN следующей конструкцией.

REFINEMENT *DiscriminatorScript*

CONSTANTS
$ext\_entrance1\,TokenType$, $extp\_entrance1\,TokenType$,
$ext\_entrance2\,TokenType$, $extp\_entrance2\,TokenType$,
$ext\_auxState1\,TokenType$, $extp\_auxState1\,TokenType$,
$ext\_auxState2\,TokenType$, $extp\_auxState2\,TokenType$,
$ext\_exitTokenType$, $extp\_exitTokenType$
PROPERTIES
$ext\_entrance1\,TokenType \in \mathbb{P}(Obj) \wedge$
$extp\_entrance1\,TokenType \in \mathbb{P}(Obj) \wedge$
$extp\_entrance1\,TokenType \subseteq ext\_entrance1\,TokenType \wedge$
$ext\_entrance2\,TokenType \in \mathbb{P}(Obj) \wedge$
$extp\_entrance2\,TokenType \in \mathbb{P}(Obj) \wedge$
$extp\_entrance2\,TokenType \subseteq ext\_entrance2\,TokenType \wedge$
$ext\_auxState1\,TokenType \in \mathbb{P}(Obj) \wedge$
$extp\_auxState1\,TokenType \in \mathbb{P}(Obj) \wedge$
$extp\_auxState1\,TokenType \subseteq ext\_auxState1\,TokenType \wedge$
$ext\_auxState2\,TokenType \in \mathbb{P}(Obj) \wedge$
$extp\_auxState2\,TokenType \in \mathbb{P}(Obj) \wedge$
$extp\_auxState2\,TokenType \subseteq ext\_auxState2\,TokenType \wedge$
$ext\_exitTokenType \in \mathbb{P}(Obj) \wedge$
$extp\_exitTokenType \in \mathbb{P}(Obj) \wedge$
$extp\_exitTokenType \subseteq ext\_exitTokenType$

VARIABLES

$entrance1$, $entrance2$, $auxState1$, $auxState2$, $exit$
INVARIANT
$entrance1 \in \mathbb{P}(ext\_entrance1\,TokenType) \wedge$
$entrance2 \in \mathbb{P}(ext\_entrance1\,TokenType) \wedge$
$auxState1 \in \mathbb{P}(ext\_auxState1\,TokenType) \wedge$
$auxState2 \in \mathbb{P}(ext\_auxState2\,TokenType) \wedge$
$exit \in \mathbb{P}(ext\_exit TokenType)$
INITIALISATION
$entrance1 := \varnothing \;||\; entrance2 := \varnothing \;||$
$auxState1 := \varnothing \;||\; auxState2 := \varnothing \;||\; exit := \varnothing$

OPERATIONS

$branch1 =$
SELECT
$\exists\, t \bullet (t \in entrance1)$
THEN
ANY t WHERE $t \in entrance1$
THEN
$entrance1 := entrance1 - \{t\} \;||$
ANY r WHERE $r \in ext\_auxState1\,TokenType$
THEN
SELECT $TRUE = TRUE$
THEN $auxState1 := auxState1 \cup \{r\}$
END
END
END
END

$branch2 =$
SELECT
$\exists\, t \bullet (t \in entrance2)$
THEN
ANY t WHERE $t \in entrance2$
THEN
$entrance2 := entrance2 - \{t\} \;||$
ANY r WHERE $r \in ext\_auxState2\,TokenType$
THEN
SELECT $TRUE = TRUE$
THEN $auxState2 := auxState2 \cup \{r\}$
END
END

END
END

trunk =
SELECT
$\exists\, t_1 \bullet (t_1 \in auxState1 \wedge auxState2 = \varnothing) \vee$
$\exists\, t_2 \bullet (t_2 \in auxState2 \wedge auxState1 = \varnothing)$
THEN
ANY t WHERE $t \in Obj \wedge$
$(t \in auxState1 \wedge auxState2 = \varnothing \vee$
$t \in auxState2 \wedge auxState1 = \varnothing)$
THEN
$auxState1 := auxState1 - \{t\} \,||\, auxState2 := auxState2 - \{t\} \,||$
ANY r WHERE $r \in ext\_exitTokenType$
THEN
SELECT $TRUE = TRUE$ THEN $exit := exit \cup \{r\}$ END
END ;
$(entrance1 := \varnothing \,||\, entrance2 := \varnothing \,||$
$auxState1 := \varnothing \,||\, auxState2 := \varnothing)$
END
END

END

Последний этап доказательства корректности расширения канонической модели рассмотренным родовым типом скрипта заключается в использовании средств автоматизации (B-Toolkit) для доказательства уточнения машины *DiscriminatorScript* машиной *Discriminator*.

Для этого необходимо согласовать машины *DiscriminatorScript* и *Discriminator*. Процесс *согласования* состоит из следующих шагов.

- Согласование направления уточнения: добавление раздела *REFINES* в машину *Discriminator*.

  REFINES *DiscriminatorScript*

- Согласование имен операций уточняемой и уточняющей машин: для каждого перехода скрипта t соответствующая операция t машины *DiscriminatorScript* переименовывается в операцию $exit\_t$; в машину *DiscriminatorScript* добавляется пустая операция $enter\_t$.

$enter\_t = skip$;

- Добавление *инварианта уточнения* в раздел *INVARIANT* машины *Discriminator*. Инвариант описывает соотношение состояний уточняющей и уточняемой машин.

$$\begin{array}{l}
\mathrm{card}\,(entrance1) = states(state_entrance1) + exec_branch1 \wedge \\
\mathrm{card}\,(entrance2) = states(state_entrance2) + exec_branch2 \wedge \\
(\,\mathrm{card}\,(auxState1) = states(state_auxState1) + exec_trunk \wedge \\
\mathrm{card}\,(auxState2) = states(state_auxState2) \vee \\
\mathrm{card}\,(auxState2) = states(state_auxState2) + exec_trunk \wedge \\
\mathrm{card}\,(auxState1) = states(state_auxState1)) \wedge \\
\mathrm{card}\,(exit) = states(state_exit)
\end{array}$$

Машины *DiscriminatorScript* и *Discriminator* в согласованном виде были введены в инструментальное средство автоматизации доказательства уточнения AMN(B-Toolkit 5.1.4). Далее автоматически были сформулированы 65 теорем, в совокупности утверждающие факт уточнения машины *DiscriminatorScript* машиной *Discriminator*. Большое количество теорем объясняется тем, что сложные теоремы автоматически разбиваются инструментальным средством на более простые теоремы, которые можно доказывать независимо. Например, теорема уточнения инициализации была разбита на 6 теорем. С использованием автоматических средств доказательства было доказано 38 теорем, остальные теоремы были доказаны интерактивно. В таблице 4.1 указано общее количество теорем и количество автоматически доказанных теорем.

5 Заключение

Настоящий период развития информационных технологий (ИТ) характеризуется взрывоподобным процессом создания разнообразных моделей представления информации. Этот процесс сопровождается другой тенденцией – накоплением использующих подобные модели информационных компонентов и сервисов, число которых экспоненциально растет. Этот рост вызывает все увеличивающуюся потребность интеграции модельно неоднородных компонентов и сервисов в различных применениях, а также их повторного использования и композиции для реализации новых информационных систем. Исследование и разработка адекватных методов оперирования разнообраз-

	Количество теорем	Количество автоматически доказанных теорем
теорема непустоты объединенного состояния	1	0
теоремы уточнения инициализации	6	6
теоремы уточнения операции *enter_branch*1	7	5
теоремы уточнения операции *exit_branch*1	7	4
теоремы уточнения операции *enter_branch*2	7	5
теоремы уточнения операции *exit_branch*2	8	5
теоремы уточнения операции *enter_trunk*	16	11
теоремы уточнения операции *exit_trunk*	13	2
общее количество теорем	65	38

Таблица 4.1. Количество теорем

ными моделями представления информации становятся чрезвычайно актуальными. Основу этих методов составляет понятие канонической информационной модели, служащей в качестве общего языка для адекватного выражения семантики разнообразных информационных моделей. В настоящем отчете суммированы описания методов синтеза канонических моделей данных, разработанных в лаборатории композиционных методов проектирования информационных систем ИПИ РАН на протяжении ряда лет. Эти методы рассматриваются отдельно для структурированных моделей данных, объектных моделей, процессных моделей. Методы синтеза канонических структурных и объектных моделей рассматриваются в обзорном плане. Это рассмотрение создает контекст для описания новых результатов – методов синтеза канонических процессных моделей, которые публикуются впервые.

Показано, что для всех трех видов моделей данных действуют одни и те же принципы приведения разнообразных моделей представления информации к унифицированному виду: принцип коммутативного отображения моделей, принцип расширения целевой модели, принцип синтеза канонической модели. Основу разработанных методов составляют понятия эквивалентности (позднее – уточнения) моделей, последовательное применение которых позволяет сохранять информацию, операции и процессное поведение при отображении исходных моделей в целевую.

Согласно принципу синтеза канонической модели, фиксируется ее ядро, конструируются расширения ядра так, чтобы они уточнялись соответствующими исходными моделями, после чего объединение всех таких расширений вместе с ядром образует каноническую модель. Процесс построения уточняющих отображений моделей данных должен быть доказательным. В работе рассмотрен широкий диапазон средств формального описания моделей и доказательства их уточнения (или эквивалентности). Так, для структурированных моделей с этой целью с успехом применялись средства денотационной семантики, а для объектных и процессных моделей – теоретико-модельные языки и средства формальных спецификаций, обеспечивающие доказательство требуемых свойств отображения моделей в логике первого порядка.

Применение методов синтеза к структурированным моделям данных было осуществлено для случая, когда в качестве ядра концептуальной модели использовалась комбинация реляционной и слабоструктурированной модели данных, а в качестве исходных моделей использовались наиболее известные структурированные модели данных. При этом техника доказательства была еще не очень развитой, доказательство осуществлялось вручную. Применение методов синтеза к объектным моделям потребовало использования теории уточнения и соответствующих инструментальных средств формального описания и автоматизированного доказательства. Те же средства были применены и для процессных моделей.

Доказательный синтез канонической модели процессов, осуществленный в данной работе, оказался возможным благодаря: 1) обнаруженной недавно возможности интерпретации одновременных событий, характерных для процессных моделей, в логике, и 2) сведению многообразия моделей потоков работ к относительно небольшому числу образцов потоков работ. Эти предпосылки послужили необходимым предусловием для синтеза канонической модели процессов, следуя провозглашенным ранее принципам.

Разработанные методы составляют необходимую основу для до-

стижения семантической интероперабельности, повторного использования и композиции неоднородных компонентов в распределенных информационных системах.

Вместе с тем, ввиду взрывоподобного развития числа и разнообразия моделей представления информации, при сохранении основных принципов отображения и синтеза моделей, применении теории уточнения, представляется затруднительным справляться вручную с таким многообразием моделей данных.

Поэтому разработанные методы должны быть дополнены компонентным подходом к синтезу канонических моделей. Этот подход в общих чертах заключается в том, что типы данных в каждой исходной модели данных регистрируются в канонической модели так, чтобы они могли служить уточнением уже введенных в каноническую модель типов или их композиций. Если таковых в существующей канонической модели не обнаруживается, следует осуществлять расширение канонической модели данных. Спецификации компонентов (типов данных) канонической и исходных моделей хранятся в репозитории, специальные инструментальные средства применяются для того, чтобы находить нужные компоненты, сопоставлять их между собой, устранять структурные и поведенческие различия компонентов, формировать их композиции, доказывать коммутативность отображений. Эту постановку планируется развить в последующем на основе имеющегося инструментария композиционного конструирования информационных систем.

Литература

1. Калиниченко Л.А. Методы и средства интеграции неоднородных баз данных. – Москва: Наука, 1983. – 423 с.

2. Калиниченко Л.А. СИНТЕЗ: язык определения, проектирования и программирования интероперабельных сред неоднородных информационных ресурсов. – Москва: ИПИ РАН, 1993.

3. Ступников С.А. Отображение канонической модели спецификаций в формальную нотацию для моделирования уточняющих спецификаций // Труды XXIV Конференции молодых ученых. – Москва, 2002. – С. 169-171.

4. W.M.P. van der Aalst, A.H.M. ter Hofstede, B. Kiepuszewski, A.P. Barros. Workflow Patterns // Distributed and Parallel Databases. – 2003. – V. 14, № 3. – P. 5-51.

5. W.M.P. van der Aalst, A.H.M. ter Hofstede. YAWL: Yet Another Workflow Language (Revised version) / Queensland University of Technology. – QUT Technical report, FIT-TR-2003-04. – Brisbane, 2003.

6. Abrial J.-R. B-Technology: Technical overview. – BP International Ltd., 1992.

7. J.-R. Abrial. The B-Book. – Cambridge University Press, 1996.

8. Jean-Raymond Abrial. B# : Toward a synthesis between Z and B // ZB'2003 - Formal Specification and Development in Z and B: Proceedings of the International Conference of B and Z Users. – 2003.

9. Backus J. Can programming be liberated from the von Neumann style? A function style and its algebra of programs // CACM. – 1978. – V. 21, № 8.

10. J. Bergstra, I. Bethke, A. Ponse. Process Algebra with Iteration and Nesting // Computer Journal. – 1994. – V. 37, № 4.

11. Briukhov D., Kalinichenko L. Component-based information systems development tool supporting the SYNTHESIS design method // Advances in Databases and Information Systems: Proceedings of the Second East European Conference. – Springer, 1998.

12. Briukhov D.O., Kalinichenko L.A., Skvortsov N.A. Information sources registration at a subject mediator as compositional

development // Advances in Databases and Information Systems: Proceedings of the 5th East European Conference. – Springer, 2001.

13. Briukhov D.O., Kalinichenko L.A., Stupnikov S.A. Compositional approach for heterogeneous sources registration at a subject mediator // Emerging Database Research in Eastern Europe: Proceedings of the Pre-Conference Workshop of VLDB 2003. – Brandenburg University of Technology at Cottbus, 2003.

14. M. Butler. csp2B: A Practical Approach to Combining CSP and B // Formal Aspects of Computing. – 2000. – V. 12.

15. CODASYL Data Description Language // Committee Journal of Development. – 1978.

16. IBM FlowMark - Modeling Workflow / IBM Corporation. – 1994.

17. C.A.R. Hoare. Communicating Sequential Processes. – Prentice-Hall, 1985.

18. K. Jensen. Coloured Petri Nets: Basic Concepts, analysis methods and practical use. – Springer, 1991.

19. Kalinichenko L.A. Data model transformation method based on axiomatic data model extension // 4th International Conference on Very Large Data Bases (VLDB): Proceedings. – 1978.

20. Kalinichenko L.A. Methods and tools for equivalent data model mapping construction // EDBT'90 Conference: Proceedings. – Springer, 1990.

21. Kalinichenko L.A. Method for Data Models Integration in the Common Paradigm // Advances in Databases and Information Systems: Proceedings of the First East-European Conference. – 1997.

22. Kalinichenko L.A. Workflow Reuse and Semantic Interoperation Issues // Advances in workflow management systems and interoperability / Edited by A.Dogac, L.Kalinichenko, M.T. Ozsu, A.Sheth. – Istanbul: NATO Advanced Study Institute, 1997.

23. Kalinichenko L.A. Compositional specification calculus for information systems development // Advances in Databases and Information Systems: Proceedings of the 3rd East European Conference. – Springer, 1999.

24. H. Ledang, J. Souquieres. Contributions for Modeling UML State-Charts in B // Integrated Formal Methods: Proceedings of the Third International Conference. – 2002.

25. MDA Guide Version 1.0.1 / OMG, document number: omg/2003-06-01. – 2003.

26. Robin Milner. Communication and Concurrency. – Prentice Hall, 1989.

27. The Object Database Standard: ODMG-93 / Edited by R.G.G. Cattell. – Morgan Kaufmann Publ., 1994. – P. 169.

28. The Common Object Request Broker: Architecture and Specification / OMG, document number 91.12.1. – 1991.

29. Petit M. Methodological clues for the design of a standard enterprise modelling language / Common Representation of Enterprise Models: Proceedings of the EI3-IC workshop. – 2002.

30. S. Schneider and J. Davies. A Brief History of Timed CSP // Theoretical Computer Science. – 1995. – V. 138.

31. H. Treharne, S. Schneider. How to Drive a B machine / Formal Specification and Development in Z and B: Proceedings of the First International Conference of Z and B Users. – 2000.

32. M. Butler, C. Snook. Verifying Dynamic Properties of UML Models by Translation to the B Language and Toolkit / Dynamic Behaviour in UML Models: Semantic Questions: Proceedings of the UML 2000 Workshop. – 2000.

33. Stupnikov S.A.,Kalinichenko L.A., Jin Song DONG. Applying CSP-like Workflow Process Specifications for their Refinement in AMN by Pre-existing Workflows / Advances in Databases and Information Systems: Proceedings of the Sixth East-European Conference. – 2002.

34. Staffware for Windows Graphical Workflow Definer / Staffware plc. – 1995.

35. Tennent R.D. The denotational semantics of programming languages // CACM. – 1976. – V. 19, №8. – P.437-453.

36. Vernadat F. B. UEML - towards a Unified Enterprise Modelling Language // MOSIM'01: Proceedings. – 2001.

37. Peter Wegner. Interaction as a Basis for Empirical Computer Science // ACM Computing Surveys. – 1995. – V. 27, № 1.

38. Peter Wegner. Why interaction is more powerful than algorithms // CACM. – 1997. – V. 40, № 5.

Научное издание

Леонид Андреевич Калиниченко
Сергей Александрович Ступников
Николай Александрович Земцов

Методы синтеза канонических моделей, предназначенных для достижения семантической интероперабельности неоднородных источников информации

Оригинал-макет: *С.А. Ступников*

Подписано в печать 00.00.00. Формат 60×90/16.
Бумага офсетная. Печать офсетная.
Усл. печ. л. 4. Уч. изд. л. 4.
Заказ № 0000. Тираж 50 экз.

Издательство ИПИ РАН
119333, г. Москва, ул. Вавилова, д. 44, корп. 2.
Лицензия ИД № 06392 от 05.12.2001.

ГУП Московская типография № 2
Федерального Агентства по печати и массовым коммуникациям.
129085, Москва, пр. Мира, 105. Тел.: 282-81-41.