Российская академия наук
Сибирское отделение
Институт систем информатики
имени А. П. Ершова

МОЛОДАЯ ИНФОРМАТИКА

Выпуск 2

СБОРНИК ТРУДОВ
АСПИРАНТОВ И МОЛОДЫХ УЧЕНЫХ

Под редакцией
к.ф.-м.н. И. С. Ануреева

Новосибирск 2006

Сборник содержит статьи, представленные аспирантами и молодыми сотрудниками ИСИ СО РАН, по следующим направлениям: теоретические аспекты программирования, информационные технологии и информационные системы, системное программное обеспечение, прикладное программное обеспечение.

© Институт систем информатики имени А. П. Ершова СО РАН, 2006

Siberian Division of the Russian Academy of Sciences
A. P. Ershov Institute of Informatics Systems

YOUNG INFORMATICS

Issue 2

COLLECTION OF PAPERS
OF GRADUATE STUDENTS
AND YOUNG SCIENTISTS

Novosibirsk 2006

The volume contains the papers presented by post-graduates and young researchers of A.P. Ershov Institute of Informatics Systems which concern the following research areas: theoretical aspects of programming, information technologies and systems, system software and application software.

© A. P. Ershov Institute of Informatics Systems, 2006

ПРЕДИСЛОВИЕ

Цель сборника — стимулирование научной деятельности аспирантов и молодых сотрудников (до 35 лет) Института систем информатики СО РАН и их обучение качественному представлению научных работ. При обучении использовалось двухэтапное рецензированием работ, и в сборник включались те статьи, которые были доработаны с учетом рецензий. Работы принимались в рамках тематики института по следующим направлениям: теоретические аспекты программирования, информационные технологии и информационные системы, системное программное обеспечение, прикладное программное обеспечение.

Целью работы «Временные структуры конфигураций, их поведенческие эквивалентности и детализация действий» является исследование зависимости поведенческих эквивалентностей от операции детализации действий в контексте временной событийно-ориентированной модели параллельных систем. Временная структура конфигураций представляет параллельную систему в виде множества ее конфигураций, предиката терминации конфигураций, помечающей и временной функций. Время непрерывно и встроено в модель таким образом, что события системы могут выполняться только в рамках сопоставленных им отрезков времени. Выполнение событий может занимать некоторый непустой промежуток времени. Вводятся понятия трассовой и сохраняющей историю бисимуляционной эквивалентностей временных структур конфигураций в семантике временных частичных порядков, и дается их сравнительная характеризация. Кроме того, вводится оператор детализации действий, используемый при проектировании параллельных систем, сопоставляющий отдельным действиям системы на данном уровне абстракции более сложные процессы на более низком уровне. Исследуется вопрос о сохранении рассматриваемых эквивалентностей временных структур конфигураций относительно операции детализации. В результате доказана инвариантность трассовой эквивалентности относительно детализации временных структур конфигураций, в отличие от сохраняющей историю бисимуляционной эквивалентности.

В работе «Человеко-машинная модель языка мышления» предлагается подход к организации языка мышления, который может использоваться при создании моделей разумного поведения. Он учитывает символьные (языковые) и несимвольные факторы и обеспечивает их взаимодействие. Вводятся понятия «абсолютный внутренний образ», «относительный внутренний образ», «внутренний символ».

Целью работы «Обнаружение взаимодействия функциональностей в телефонных сетях с помощью раскрашенных сетей Петри» является выявление взаимодействия функциональностей в моделях телефонных сетей, представленных раскрашенными сетями Петри (РСП), при помощи программной верификации.

Для программной верификации используется система SPV. Для каждой РСП строится граф достижимости, используемый при верификации свойств, представленных формулами μ-исчисления, методом проверки моделей.

В основе любой телефонной сети лежит так называемая «базовая модель звонков» (basic call state model). Для обнаружения взаимодействия функциональностей телефонных сетей построена РСП для базовой модели звонков и 5 функциональностей. Программной верификацией обнаружены их нежелательные взаимодействия.

Система UniCalc представляет собой многофункциональную среду для решения задач математического моделирования с удобным графическим интерфейсом. Математический аппарат системы UniCalc основан на недоопределенных вычислениях, позволяющих находить внешнюю оценку для множества решений произвольной системы ограничений.

В работе «Трехмерная визуализация множества решений в системе UniCalc» описан графический модуль, предоставляющий пользователю дополнительные средства анализа математических моделей. Суть работы модуля состоит в графическом отображении множества значений переменных, удовлетворяющих заданной системе ограничений.

График строится постепенной детализацией покрытия, где покрытие — множество параллелепипедов, содержащее все точки реального графика. Первым покрытием графика является параллелепипед, образованный интервалами значений всех переменных. Затем выполняется пошаговая детализация: на каждом шаге из покрытия удаляется один параллелепипед и, если он может содержать точки графика, то он разбивается на 8 равных параллелепипедов (в двухмерном случае на 4), которые добавляются в покрытие. Этот процесс продолжается до тех пор, пока не будет достигнут нужный уровень детализации.

Наличие решений у заданной системы ограничений в некоторой области пространства (в нашем случае, параллелепипеде) определяется с помощью недоопределенных вычислений. Получение внешней оценки, равной пустому множеству, означает, что заданная система ограничений и ограничения, задающие параллелепипед, несовместны, т.е. параллелепипед не содержит точек графика.

Преимуществами такого способа визуализации являются своевременный отсев областей пространства, заведомо не содержащих решений системы, и корректность отображения множества решений (на каждом шаге покрытие гарантированно содержит все решения системы).

В модуле реализованы возможности двухмерной и трехмерной визуализации.

Трехмерная графика выполнена в проволочной модели на основе алгоритма отсечения по пирамиде видимости, благодаря которому легко реализуются изменение угла обзора, масштаба. Разработанный графический модуль позволяет выбирать размерность визуализации, переменные осей абсцисс, ординат и аппликат, управлять настройками графиков (числом, порядком, цветом), пространством обзора, а также предоставляет другие возможности по работе с графикой.

Унифицированный язык моделирования UML является фактически стандартом в сфере промышленного производства программного обеспечения. Тем не менее, UML не лишен проблем, связанных с неформальной нотацией языка, и, как следствие, возникает неоднозначная трактовка моделируемых систем.

В работе «Формальная модель диаграммы классов языка UML» рассматривается формальная модель подмножества языка UML — диаграммы классов. При построении модели использованы простейшие понятия теории множеств и многоосновных алгебр. Основой подхода является использование машин абстрактных состояний (МАС) Ю. Гуревича.

Моделирование и верификация выполнимых спецификаций, представленных на языке SDL, является одной из проблем современного программирования. Подход к этой проблеме, который исследуется в лаборатории теоретического программирования ИСИ СО РАН, состоит в разработке модельных языков, ориентированных на их верификацию. Таким языком является разработанный лабораторией комбинированный язык спецификаций REAL, который используется в качестве промежуточного языка в системе верификации SDL-спецификаций. Транслятор из языка спецификаций SDL в язык REAL является важной частью этой системы.

Целью работы «Трансляция языка выполнимых спецификаций распределенных систем SDL в язык выполнимых спецификаций REAL» является описание системы трансляции выразительного подмножества языка SDL в язык выполнимых спецификаций REAL.

Для преодоления трудностей программной реализации был разработан двухпроходной процесс трансляции и специальное внутреннее представление.

В качестве входного языка системы трансляции выбрано представительное подмножество языка SDL88, которое включает его динамические конструкции. Выходным языком системы трансляции является язык Dynamic-REAL. Описанный в работе транслятор по SDL-спецификации эффективно строит эквивалентную спецификацию на языке Dynamic-REAL.

Система реализована на языке C++. Синтаксический анализатор построен генератором синтаксических анализаторов BISON.

Данная система трансляции объединена с системой верификации REAL спецификаций.

Многие крупные организации сегодня продолжают использовать приложения, написанные на языках типа Cobol, PL/I, Natural и др. Сопровождение таких систем требует существенных затрат, поскольку для каждого даже небольшого изменения кода необходим тщательный предварительный анализ. Следует заметить, что восстановление бизнес-логики приложения более эффективно, чем многократный частичный анализ кода. Однако восстановление бизнес-логики реального приложения вручную – достаточно трудоемкий и длительный процесс. В работе «Автоматическое восстановление бизнес-логики программ» описывается функциональность AutoDetect системы Modernization Workbench, частично автоматизирующая этот процесс. AutoDetect позволяет автоматически строить бизнес-правила, основываясь на информационном графе программы.

Для указанной переменной в коде AutoDetect находит все операторы, участвующие в ее вычислении. Каждому оператору сопоставляется бизнес-правило. Бизнес-правила располагаются в порядке, соответствующем правильному порядку исполнения операторов. К каждому бизнес-правилу пользователь может добавить собственные комментарии. Таким образом, предлагаемая функциональность способствует пониманию кода и может использоваться для документирования приложений.

Открытые морфизмы активно используются для характеризации различных эквивалентностей для параллельных систем и процессов. Работа «Открытые морфизмы и временная тестовая эквивалентность для временных автоматных моделей» посвящена исследованию временного варианта тестовой эквивалентности в контексте временных автоматных моделей. В качестве модели были выбраны временные системы переходов, которые, по сути, являются обычными временными автоматами без множества поглощающих состояний и условий принятия. В частности, в работе была определена категория временных систем переходов CTTStest и выделена ее подкатегория Ptest. Кроме того, было показано, что построенная категория

и ее подкатегория обладают всеми необходимыми свойствами для применения методов теории категорий. Далее, следуя стандартной схеме, предложенной Винскелем, Нильсеном и Джойялем, было введено понятие открытого морфизма, основанное на подкатегории Ptest, и доказан критерий открытости для морфизмов. В заключение была определена абстрактная бисимуляция в терминах существования конструкции открытых морфизмов и доказано, что эта бисимуляция совпадает с временным вариантом тестовой эквивалентности.

В работе «Разработка модели адаптивного поведения анимата на основе семантического вероятностного вывода» предложена адаптивная система управления аниматом (искусственным организмом), основанная на семантическом вероятностном выводе и теории функциональных систем П.К. Анохина. Система управления строится на основе иерархии функциональных систем, формирующихся для достижения полезных для анимата целей. Семантический вероятностный вывод используется для нахождения наиболее оптимальных способов достижения цели. Основным отличием данной модели является возможность автоматического выявления новых подцелей. Основываясь на предложенной модели, был построен анимат и проведен ряд экспериментов по его обучению и сравнению с существующими подходами, основанными на нейронных сетях и потактовом обучении (Reinforcement Learning). Результаты сравнения показали, что предложенная модель обучается и действует эффективнее.

В последние годы все большее распространение получают базы данных в формате XML. Наиболее перспективным языком запросов для них считается XQuery. При обработке запросов к базе данных выражение на языке запроса обычно транслируется в выражение соответствующей алгебры и затем производится вычисление полученного выражения. Алгебра задает семантику языка запросов и обычно поддерживает набор правил оптимизации полученного в результате трансляции выражения. Имеется достаточно много статей, представляющих различные алгебры. Одним из главных недостатков этих алгебр является то, что рассматривается лишь синтаксический перевод выражения XQuery на язык алгебры. Семантика запроса при таком переводе может не сохраниться, и в результате становится сложно формально доказать правильность полученных значений.

Алгебра А.В. Замулина выбрана за основу в работе «XML-алгебра для языка запросов XQuery» как алгебра, сохраняющая семантику запроса. Одним из понятий, не включенных в эту алгебру, является понятие пространства имен. Имя объекта базы данных локально в некотором пространстве имен. Разные объекты, принадлежащие разным пространствам имен, могут

иметь совпадающие имена. Идентификация объектов реализуется по полному имени, состоящему из имени пространства имен и локального имени. Структура базы данных задается множеством XML-схем, которые могут иметь совпадающие имена атрибутов или элементов. Предлагаемая работа содержит в себе краткий обзор существующих алгебр и расширение алгебры А.В. Замулина для возможности работы с разными пространствами имен.

В работе «Формальная модель основных понятий языка C#» представлена формальная семантика большинства типичных понятий языка C#. К ним относятся: система типов (примитивные типы, классы, структуры, интерфейсы и делегаты), свойства, события, индексаторы, отношения наследования и реализации, механизм совмещения имен и подмены методов и др.

Состояние программы определяется как многоосновная алгебра, компоненты которой один в один соответствуют состоянию памяти реальной программы и реальным операциям по манипулированию ячейками памяти. Тогда функции программы представляются математическими функциями, анализирующими и/или преобразующими состояние программы. Таким образом, программу, записанную на языке C#, мы будем рассматривать как представление аналитического выражения в подходящей алгебре. В некотором смысле соблюдается стиль алгебраических спецификаций, в которых сначала определяется некоторая сигнатура, затем определяется множество ее моделей, затем определяется, как конструируются термы данной сигнатуры и как они интерпретируются в данной модели и, наконец, конструируется спецификация и определяется, что является множеством ее моделей.

PREFACE

The purpose of the volume is to stimulate research activity of post-graduates and young researchers of A.P. Ershov Institute of Informatics Systems and to train them in qualitative presentation of scientific papers. Training has been based on two-stage paper reviewing, and the volume contains only those papers which were improved according to reviewers' remarks. To be accepted, the paper should cover one of the research topics of the Institute, such as theoretical aspects of programming, information technologies and systems, system software and application software.

The paper "Timed Configuration Structures: Equivalence Notions and Action Refinement" studies the interplay between the equivalence notions and action refinement in the setting of a real-time event-oriented model of a concurrent system. A configuration structure represents a system by a set of its configurations, a termination predicate, and labeling and timing functions. Time is incorporated into the configuration structures in such a way that system events can occur exactly during the corresponding timed intervals and the occurrence of each event can take some time. The notions of trace equivalence and history preserving bisimulation for timed configuration structures are studied in terms of timed partial orders. An operator for refinement of actions is intended to be used in the design of concurrent models. It substitutes actions at the given level of abstraction by more complicated processes at the lower level. A question of preservation of these equivalences under refinement of timed configuration structures is investigated. Finally, it is shown that the trace equivalence is invariant under refinement, but it is not the case for history preserving bisimulation.

The paper "Human-Machine Model of Language of Thinking" presents an approach to organization of a language of thinking which may be used for creation of models of intellectual behavior. It takes into account both symbolic (linguistic) and non-symbolic factors and provides their interaction. The concepts of an absolute internal image, a relative internal image and an internal symbol are introduced.

The paper "Detection of feature interaction in telephone networks using colored Petri nets" shows how program verification can be used for feature interaction detection in telephone networks modeled by coloured Petri nets (CPN). For program verification, the SPV system is used. A reachability graph is built for every CPN and it is used for verification of the system properties represented by mu-calculus formulae using the model checking method.

Every telephone network is based on the so-called Basic Call State Model (BCSM). For detection of feature interaction in a telephone network, a CPN model for BCSM with 5 features is built. Some unwanted feature interactions have been detected using program verification.

The UniCalc system is a powerful environment with convenient graphical interface for solving the mathematical modeling problems. The core solver of the UniCalc system implements subdefinite calculations and finds a set containing all solutions for an arbitrary system of constraints.

A graphical module which provides a user with additional tools for analysis of mathematical models is described in the paper "3D visualization of a solution set in the UniCalc system". In a graphic form, it represents the set of variable values that satisfy the system of constraints.

The graph is constructed by stepwise refinement of a covering which is a set of parallelepipeds containing all points of a real graph. The first covering is a parallelepiped formed by intervals of all variable values. Then the stepwise refinement is performed: at each step one parallelepiped is removed from the covering and, if it contains any point of the real graph, it is divided into 4 or 8 equal smaller parallelepipeds which are added to the covering. The process goes on till the needed level of detail is reached.

Solvability of the given set of constraints in some domain (parallelepiped) is determined with the help of subdefinite calculations. If we obtain an empty set as an external estimate, then the given system and constraints from the parallelepiped are incompatible, i.e. this parallelepiped does not contain any point of the real graph.

This method of visualization allows us to exclude empty parallelepipeds from the covering and to provide a reliable representation of the solution set.

2D- and 3D-visualization are implemented in this module. 3D-graphics is based on the algorithm of viewing frustum, so any turn and scaling can be easily performed. Thus, a user can choose the dimension of visualization and the variables for abscissa, ordinate and applicate, control the graph properties and viewing space, etc.

The Unified Modeling Language (UML) is de facto a standard of the software development industry. Nevertheless, UML has some problems with its nonformal notation.

In the paper "Formal model of the UML class diagram", a formal model of a subset of the UML language — class diagrams — is described. The model is developed with the use of notions of the set theory and polybasic algebra. This approach is based on the evolving algebra by Yu. Gurevich.

Simulation and verification of executable specifications of distributed systems represented in SDL is a topical research problem. An approach to this problem which is being developed in the IIS laboratory of theoretical programming is based on the development of model languages oriented to their verification. Such a language is REAL used as an intermediate language in a system of simulation and verification of SDL specifications. The translator from SDL to REAL is an important part of the system. It is described in the paper "Translation of a language of executable specifications of distributed systems SDL into a language of executable specifications REAL".

To overcome the difficulties of implementation, a double-line process of translation and a special internal structure have been developed.

A representative subset of SDL88 including its dynamic constructions was selected as an input language. The target language of the system of translation is Dynamic-REAL.

The system is implemented in C++. The syntactical analyzer is built with the help of the generator of syntactical analyzers BISON.

Today many big companies are still working with legacy applications implemented in the programming languages like COBOL, PL/I, Natural, etc. Maintenance of such systems requires a large amount of resources, because thorough analysis is needed before every even a little change in the code. Note that recovery of business logic of a program is more efficient than repeated partial code analysis.

However, the manual recovery of business logic is rather time-consuming and difficult for real applications. The AutoDetect functionality of the Modernization Workbench system that partially automates the process is described in the paper "Automatic recovery of program business logic". AutoDetect automatically creates business rules for a program using its information graph.

For any selected variable in a code, AutoDetect finds all statements needed for computation of this variable and a business rule is created for each statement. Business rules are arranged in the order corresponding to the correct statement execution order. A user can add his comments to each business rule. Therefore the described functionality is useful for both program understanding and application documenting.

Open maps have been actively used for characterization of different equivalences of concurrent systems and processes. The paper "Open maps and timed testing equivalence for timed automata models" presents an investigation of a timed variant of testing equivalence in the setting of timed automata models. Timed transition systems, used as a model, are basically timed automata without a set of accepting states and acceptance conditions. In particular, this paper de-

fines a category of timed transition systems CTTStest and the path subcategory Ptest and they are shown to have the required properties for applying the category-theoretic approach. Next, using the general framework proposed by Winskel, Nielsen and Joyal, a notion of open maps based on the subcategory Ptest is obtained and a criterion of openness for morphisms is established. Finally, an abstract bisimulation in terms of the existence of a span of open maps is defined and it is shown that this one is equivalent to timed testing equivalence.

The model of adaptive behavior of the autonomous adaptive agents (artificial organism) based on the semantic probabilistic inference and the functional system theory by P.K. Anokhin is presented in the paper “The model of adaptive behavior based on the semantic probabilistic inference”. The control system is based on the hierarchy of functional systems that were formed in order to achieve some special purposes useful for the animat. The semantic probabilistic inference is used for finding the optimal ways of purpose achievement. The main advantage of the model is the possibility to form new purposes. Based on this model, an autonomous adaptive agent has been developed and experiments have been performed on its learning and functioning in comparison with other models based on reinforcement learning and neural networks. The results show that its actions are more efficient.

XML database systems have recently become very popular. XQuery is regarded to be the most perspective query language for these systems. To process a database system’s query, the query expression is usually translated to an expression in the corresponding algebra in order to compute it. The algebra reflects semantics of the query language and usually supports a set of optimization rules for the translated expression. There are a lot of articles that represent different algebras. One of the main disadvantages of these algebras is that only syntactical translation of XQuery expression to the algebraic language is performed. In this translation, semantics of a query may be lost and, as a result, accuracy of values so obtained is become difficult to prove formally.

The algebra proposed by A.V. Zamulin is chosen as a basis in the paper “XML Algebra for XQuery” since it keeps the query semantics. One of the notions not included in this algebra is the notion of a namespace. The name of a database object is local in some namespace. Different objects from different namespaces may have the same names. Identification of an object is made by its full name consisted of the namespace name and a local name. The database system structure is specified by XML schemas some of which may have coincident names of attributes or elements. This work encloses a brief review of present algebras and describes an expansion of A.V. Zamulin’s algebra with the namespace notion.

Formal semantics of the basic notions of the C# language is presented in the paper “Formal model of the basic concepts of the programming language C#”. These notions are the following: a type system (basic types, classes, structures, interfaces, and delegates), properties, events, indexers, relations of inheritance and implementation, name aliasing and method substitution mechanisms, etc.

A program state is defined to be a many sorted algebra such that there is one-to-one correspondence between its components and real program memory states and memory-modifying operations. So, program functions are represented via mathematical functions that analyze and/or modify a program state. Thus, a program written in the C# language is considered as an analytical expression in a suitable algebra. This corresponds to the algebraic specification style – first, a signature is defined, second, a set of its models is defined, third, terms are constructed and their interpretation is defined, and finally, the specification is constructed and the set of its models is defined.

М.В. Андреева

ВРЕМЕННЫЕ СТРУКТУРЫ КОНФИГУРАЦИЙ: ПОВЕДЕНЧЕСКИЕ ЭКВИВАЛЕНТНОСТИ И ДЕТАЛИЗАЦИЯ ДЕЙСТВИЙ

1. ВВЕДЕНИЕ

На протяжении последних лет было предложено множество разнообразных эквивалентностей параллельных систем, взаимосвязи между которыми были широко изучены в литературе [7]. Можно выделить два основных критерия классификации эквивалентностных понятий: по степени, с которой эквивалентности учитывают детали вычисления систем, и по степени, с которой они учитывают состояния выбора системой между возможными дальнейшими вычислениями. Крайними представителями первого критерия являются интерливинговая семантика и семантика частичных порядков. В интерливинговой семантике [11] выполнение системы моделируется последовательностью действий, не отражающей явно их причинную зависимость. В семантике частичных порядков [7, 14] все причинные связи между выполняемыми действиями системы сохраняются, что позволяет моделировать отношение параллелизма явным образом. Простейшим представителем второго критерия является не учитывающая ветвящейся структуры выбора трассовая семантика [11], в которой поведение системы представлено множеством ее возможных выполнений. С другой стороны спектра находится бисимуляционная семантика [1, 7], строго учитывающая точки ветвления различных вычислений системы.

При проектировании параллельных систем часто используется операция детализации действий, позволяющая представлять поведение системы на более высоком или более низком уровнях абстракции. Оператор сопоставляет действиям на данном уровне абстракции более сложные процессы на более низком уровне. В ряде работ [6, 7, 9] исследовалась проблема инвариантности упомянутых выше поведенческих эквивалентностей параллельных систем относительно детализации действий. В частности, было установлено, что из эквивалентностей, соответствующих первому критерию, при детализации действий сохраняются только представители семантики частичных, в отличие, например, от интерливинговой семантики.

В последние годы в литературе наблюдается растущий интерес к моделированию систем реального времени, вследствие которого возникает необходимость в формальном выражении течения времени. Для таких систем было предложено несколько формальных методов спецификации и их обоснований [2, 3]. В нескольких работах [5, 13, 18] исследовался ряд вопросов, относящихся к эквивалентностным понятиям, учитывающим ход времени. В этих исследованиях системы реального времени были представлены временными интерливинговыми моделями — процессами с параллельными таймерами или временными автоматами, содержащими фиктивные элементы, отмеряющие время, и называемые часами.

В данной работе исследуется вопрос о сохранении поведенческих эквивалентностей после применения операции детализации действий в контексте временных структур конфигураций с предикатом терминации [9], позволяющим различать успешно завершенные вычисления системы (successful termination) и тупиковые (deadlock). В частности, рассматриваются временные расширения трассовой эквивалентности и сохраняющей историю бисимуляции в семантике временных частичных порядков. Встроенное в модель время непрерывно и глобально, но в отличие от [4, 15], выполнение событий не мгновенно, а имеет некоторую длительность, представленную временными интервалами.

Оставшаяся часть работы распределена следующим образом. В разд. 2 напоминаются основные понятия и обозначения теории структур конфигураций. В разд. 3 вводятся временные структуры конфигураций и операция детализации действий на них. Далее, в разд. 4 рассматриваются понятия трассовой эквивалентности и сохраняющей историю бисимуляции временных структур конфигураций и разрешается вопрос об их инвариантности относительно операции детализации действий. Разд. 5 содержит заключение, а также замечания по поводу будущих работ.

2. СТРУКТУРЫ КОНФИГУРАЦИЙ И ДЕТАЛИЗАЦИЯ ДЕЙСТВИЙ

В данном разделе мы напоминаем основные понятия теории структур конфигураций [8-10], которые являются обобщением моделей структур событий [16, 17]. Структура конфигураций представляет систему в виде множества ее конфигураций, предиката терминации [9] и помечающей функции. Конфигурации представляют отдельные вычисления системы, состоящие из множества выполнившихся событий. Предикат терминации позволяет определить, какие из максимальных конфигураций соответству-

ют успешно завершенным вычислениям (successful termination), а какие — тупикам (deadlock). Помечающая функция сопоставляет каждому событию определенное действие.

Определение 2.1. Пусть Act — конечное множество действий, **E** — счетное множество событий. *(Помеченной) структурой конфигураций* (*над Act*) называется набор $S = (Conf, \surd, l)$, где

- $Conf \subseteq \mathcal{P}_F(\mathbf{E})$ — семейство конечных подмножеств событий (*множество конфигураций*);
- $\surd \subseteq Conf$ — *предикат терминации,* удовлетворяющий условию $C \subseteq \surd \wedge C \subseteq C_1 \Rightarrow C = C_1$;
- $l : E_S = \bigcup_{C \in Conf} C \rightarrow Act$ — помечающая функция, сопоставляющая событиям конфигураций структуры действия из Act.

Множество структур конфигураций обозначим через $\mathcal{S}$.

На событиях конфигураций естественным образом формируется отношение причинной зависимости.

Определение 2.2. Пусть S — структура конфигураций, $C \in Conf_S$. Тогда *отношение причинной зависимости* $\leq_C$ на событиях в C определяется следующим образом: $e_1 \leq_C e_2 \Leftrightarrow \forall C_1 \subseteq C \diamond e_2 \in C_1 \Rightarrow e_1 \in C_1$.

Для построения семантики частичного порядка, далее нам понадобится следующее определение.

Определение 2.3. Структура конфигураций $S = (Conf, \surd, l)$ называется

- *имеющей корень*, если $\varnothing \in Conf$;
- *связной,* если $\varnothing \neq C \in Conf \Rightarrow \exists e \in C \diamond C \setminus \{e\} \in Conf$;
- *замкнутой относительно конечного объединения,* если $C_1, C_2, C_3 \in Conf \wedge C_1 \cup C_2 \subseteq C_3 \Rightarrow C_1 \cup C_2 \in Conf$;
- *замкнутой относительно конечного пересечения,* если $C_1, C_2, C_3 \in Conf \wedge C_1 \cup C_2 \subseteq C_3 \Rightarrow C_1 \cap C_2 \in Conf$.

S называется *стабильной*, если она имеет корень, связна и замкнута относительно конечного объединения и пересечения.

Теорема 2.1. [9] Структура конфигураций S стабильна тогда и только тогда, когда для всех $C \in Conf_S$ верно:

I. $\leq_C$ является частичным порядком;
II. $C_1 \in Conf_S \Leftrightarrow C_1$ левозамкнуто относительно $\leq_C$ для всех $C_1 \subseteq C$.

Далее мы будем рассматривать только стабильные структуры конфигураций, называя их просто структурами конфигураций.

Пример 2.1. Рассмотрим простейший пример структуры конфигураций $S = (Conf, \surd, l)$, где $Conf = \{\emptyset, \{e_1\}, \{e_1,e_2\}\}$, $\surd = \{\{e_1,e_2\}\}$ и $l(e_1) = a$, $l(e_2) = b$. S стабильна, $E_S = \{e_1,e_2\}$ и $e_1 \leq_C e_2$, где $C = \{e_1,e_2\}$.

Операция детализации структур конфигураций состоит в замещении событий конфигураций на структуры конфигураций, соответствующие помечающим события действиям [9]. Мы рассматриваем детализацию действий, запрещающую "забывать" события, т. е. замещать их на структуру конфигураций $\varepsilon = (\{\emptyset\},\{\emptyset\},\emptyset)$, соответствующую успешно завершенному процессу, не выполнившему ни одного действия.

Определение 2.4. Пусть $S = (Conf, \surd, l)$ — структура конфигураций и *ref* — функция детализации.

- Функция $ref : Act \rightarrow \mathcal{S} \setminus \{\varepsilon\}$ называется *функцией детализации структур конфигураций*.
- Назовем $\check{C}$ *детализацией конфигурации* $C \in Conf$ *с помощью ref*, если
  - $\check{C} = \bigcup_{e \in C} \{e\} \times C_e$, где $\forall e \in C \diamond C_e \in Conf_{ref(l(e))} \setminus \{\emptyset\}$;
  - $\forall X \subseteq busy(\check{C}) \diamond C \setminus X \in Conf$, где $busy(\check{C}) := \{e \in C \mid C_e \notin \surd_{ref(l(e))}\}$.

  Такую детализацию $\check{C}$ назовем *терминированной*, если $busy(\check{C}) = \emptyset$.
- Определим *детализацию* S *с помощью ref* как $ref(S) := (C_{ref(S)}, \surd_{ref(S)}, l_{ref(S)})$, где
  - $Conf_{ref(S)} := \{ \check{C} \mid \check{C}$ — детализация некоторой $C \in Conf$ с помощью $ref \}$;
  - $\surd_{ref(S)} := \{ \check{C} \mid \check{C}$ — терминированная детализация некоторой $C \in \surd$ с помощью $ref \}$;
  - $l_{ref(S)}(e,e') := l_{ref(l(e))}(e)$ для всех $(e,e') \in E_{ref(S)}$.

Как было показано в [9], детализация $ref(S)$ структуры конфигураций S с помощью функции детализации *ref* также является структурой конфигураций. Более того, если S имеет корень, связна, замкнута относительно конечного объединения или пересечения, или стабильна, то $ref(S)$ наследует эти свойства. Также было показано, что если $\check{C} \in Conf_{ref(S)}$ детализирует $C \in Conf_S$, то $(e_1,e_1') <_{\check{C}} (e_2,e_2') \Leftrightarrow e_1 <_C e_2 \vee (e_1 = e_2 \wedge e_1' <_{C_e} e_2')$.

3. ВРЕМЕННЫЕ СТРУКТУРЫ КОНФИГУРАЦИЙ И ДЕТАЛИЗАЦИЯ ДЕЙСТВИЙ

В данном разделе вводится непрерывно-временное расширение структур конфигураций, или *временные структуры конфигураций.* Предполагается глобальный непрерывный счетчик времени, течение которого не отделено от выполнения событий. В отличие от [4, 15], выполнение событий не мгновенно, а имеет длительность — конечный промежуток времени, ограниченный соответствующими временными рамками.

Пусть R — множество неотрицательных действительных чисел. Обозначим множество отрезков в R через $Interv := \{ [d_1,d_2] \subset \mathrm{R} \mid d_1 \le d_2 \}$.

Определение 3.1. *(Помеченной) временной структурой конфигураций* (*над Act*) называется набор $TS = (S, D)$, где

- $S := (Conf, \surd, l)$ — структура конфигураций;
- $D : E_S \to Interv$ — функция, сопоставляющая событиям *временные интервалы*, в рамках которых они могут выполниться.

Обозначим множество временных структур конфигураций через $\mathcal{TS}$.

Временные структуры конфигураций моделируют отдельные вычисления параллельных систем с помощью временных конфигураций. Временная конфигурация состоит из конфигурации (множества выполнившихся событий) и временной функции, сохраняющей отрезки времени, в течение которых выполнялись события.

Определение 3.2. Пусть $TS = (S = (Conf, \surd, l), D)$ — временная структура конфигураций, $C \in Conf$ и $T : C \to Interv$.

Тогда $TC = (C,T)$ — *временная конфигурация* в TS, если

- $T(e) \subseteq D(e)$ для всех $e \in C$;
- $\max T(e_1) \le \min T(e_2)$, если $e_1 \le_C e_2$.

Множество временных конфигураций в TS обозначим через $TConf$.

Будем говорить, что временная конфигурация $TC_1 = (C_1,T_1)$ *переходит* во временную конфигурацию $TC_2 = (C_2,T_2)$, обозначается $TC_1 \to TC_2$, если $C_1 \subseteq C_2$ и $T_2|_{C_1} = T_1$.

Назовем $TS = (S,D) \in \mathcal{TS}$ *корректно таймированной*, если $e_1 \le_C e_2$ влечет $\min D(e_1) \le \min D(e_2)$ и $\max D(e_1) \le \max D(e_2)$ для всех $C \in Conf_{TS}$. Преимущество корректного таймирования состоит в том, что для любой конфигурации существует временная функция, составляющая с ней временную конфигурацию. Более того, любое «начало» временной функции, опреде-

ленное на подконфигурации, может быть продолжено на всю конфигурацию.

Лемма 3.1. Пусть $TS \in \mathcal{TS}$ — корректно таймированная, $TC = (C,T) \in TConf_{TS}$ и $C \subseteq C_1 \in Conf_S$.

Тогда существует функция $T_1 : C_1 \to Interv$ такая, что $TC_1 = (C_1,T_1) \in TConf_{TS}$ и $TC \to TC_1$.

Доказательство следует из определений отношения перехода и корректного таймирования. □

Далее мы будем рассматривать только корректно таймированные временные структуры конфигураций, называя их просто временными структурами конфигураций.

Пример 3.1. Рассмотрим временную структуру конфигураций $TS = (S,D)$, где $S = (\{\emptyset,\{e_1\},\{e_1,e_2\}\}, \{\{e_1,e_2\}\}, \{(e_1,a),(e_2,b)\})$ из примера 2.1, и $D(e_1)= [0,2]$, $D(e_2) = [1,3]$. TS корректно таймирована и $TConf_{TS} = \{(\emptyset,\emptyset), (\{e_1\}, T_1), (\{e_1,e_2\}, T_2) \mid T_1 (e_1) = T_2 (e_1) = [d_1,d_1'] \subseteq [0,2] , T_2 (e_2) = [d_2,d_2'] \subseteq [d_1', \infty] \cap [1,3]\}$.

Теперь мы можем определить операцию детализации временных структур конфигураций. Как и в случае структур конфигураций, детализация действий состоит в замещении событий конфигураций на структуры конфигураций, соответствующие помечающим события действиям [9], при этом предполагается наследование временных рамок исходных событий событиями замещающих структур конфигураций.

Определение 3.3. Пусть $TS = (S,D) \in \mathcal{TS}$, и ref — функция детализаций. Тогда $ref(TS) := (ref(S), D_{ref(TS)})$ — *детализация* TS *с помощью* ref, где $D_{ref(TS)} (e,e') := D(e)$ для всех $(e,e') \in E_{ref(S)}$.

Легко проверить, что детализация временной структуры конфигураций также является временной структурой конфигураций. Далее рассмотрим понятие детализации временной конфигурации.

Определение 3.4. Пусть $TS \in \mathcal{TS}$ и ref — функция детализации. Назовем $\check{T}\check{C} = (\check{T},\check{C})$ *детализацией временной конфигурации* $TC = (T,C) \in TConf_{TS}$ *с помощью* ref, если $\check{C}$ является детализацией C и

$$\check{C} = \bigcup_{e\in C} \{e\}\times C_e \quad \text{и} \quad \check{T}(e,e') = T_e(e') \quad \text{для всех } (e,e') \in \check{C},$$

где $TC_e = (T_e, C_e) \in TConf_{TS_e} \setminus \{(\varnothing,\varnothing)\}$, $TS_e = (ref(l_{TS}(e)), D_e) \in \mathcal{TS}$ и $D_e(e') = T(e)$ для всех $e' \in E_{TS_e}$.

Таким образом, детализация временной конфигурации состоит из объединения временных конфигураций, детализирующих временные события исходной временной конфигурации. Следующая теорема говорит о том, что каждая такая детализация временной конфигурации также является временной конфигурацией, и их множество совпадает с множеством временных конфигураций в детализированной временной структуре конфигураций.

Теорема 3.1. Пусть $TS = (S,D) \in \mathcal{TS}$ и ref — функция детализации, тогда $TConf_{ref(TS)} = \{\check{T}\check{C} \mid \check{T}\check{C}$ — детализация некоторой $TC \in TConf_{TS}$ с помощью $ref\}$.

Доказательство

$(\subseteq)$

Пусть $\check{T}\check{C} = (\check{T},\check{C}) \in TConf_{ref(TS)}$. Тогда $\check{C} \in Conf_{ref(S)}$ детализирует некоторую $C \in Conf_S$ с помощью ref, т. е.

$$\check{C} = \bigcup_{e \in C} \{e\} \times C_e, \text{ где } \forall e \in C \diamond C_e \in Conf_{ref(l_S(e))} \setminus \{\varnothing\}.$$

Построим временные функции $T : C \to Interv$ и $T_e : C_e \to Interv$ для всех $e \in C$, где

$$T(e) := [\min \bigcup_{e' \in C_e} \check{T}(e,e'),\ \max \bigcup_{e' \in C_e} \check{T}(e,e')] \text{ и}$$
$$T_e(e') := \check{T}(e,e') \text{ для всех } e \in C \text{ и } e' \in C_e.$$

Теперь $\check{T}$ можно представить как $\check{T}(e,e') = T_e(e')$ для всех $(e,e') \in \check{C}$. Для всех $e \in C$ построим $TS_e = (ref(l_S(e)), D_e)$, где $D_e(e') = T(e)$ для всех $e' \in E_{TS_e}$. Покажем, что $\check{T}\check{C}$ детализирует $TC = (C,T) \in TConf_{TS}$.

1) $TC = (C,T) \in TConf_{TS}$:
   так как $\check{T}\check{C} \in TConf_{ref(TS)}$, то $\check{T}(e,e') \subseteq D_{ref(TS)}(e,e') = D_{TS}(e)$ для всех $(e,e') \in \check{C}$. Тогда $T(e) \subseteq D_{TS}(e)$ для всех $e \in C$. Кроме того, если $e_1 <_C e_2$, то $(e_1, e_1') <_{\check{C}} (e_2, e_2')$ и по определению временной конфигурации $\max \check{T}(e_1,e_1') \le \min \check{T}(e_2,e_2')$ для всех $e_1' \in C_{e_1}$ и $e_2' \in C_{e_2}$, то $\max T(e_1) \le \min T(e_2)$;
2) очевидно, что $TS_e = (ref(l(e)), D_e) \in \mathcal{TS}$ для всех $e \in C$;
3) $TC_e = (T_e, C_e) \in TConf_{TS_e}$:
   по построению, $T_e(e') \subseteq D_e(e')$ для всех $e' \in C_e$.
   Для $e' <_{C_e} e''$ получаем $(e,e') <_{\check{C}} (e,e'')$, что влечет
   $\max \check{T}(e,e') \le \min \check{T}(e,e'')$, или $\max T_e(e') \le \min T_e(e'')$.

$(\supseteq)$

Пусть $\check{T}\check{C}$ — детализация некоторой $TC \in TConf_{TS}$ с помощью *ref*. Покажем, что $\check{T}\check{C} = (\check{T},\check{C}) \in TConf_{ref(TS)}$.

Так как $\check{C} \in Conf_{ref(S)}$, проверим, что $\check{T}$ удовлетворяет требованиям определения. По условию, $\check{T}(e,e') \subseteq T(e) \subseteq D_{TS}(e) = D_{ref(TS)}(e,e')$ для всех $(e,e') \in \check{C}$. Для $(e_1, e_1') <_{\check{C}} (e_2, e_2')$ возможны следующие два случая:

1) если $e_1 = e_2$, то $e_1', e_2' \in C_e$ и $e_1' <_{C_e} e_2'$, что влечет
   $\max T_e(e_1') \le \min T_e(e_2')$, или $\check{T}(e_1,e_1') \le \min \check{T}(e_2,e_2')$;
2) если $e_1 <_C e_2$, то $\max T(e_1) \le \min T(e_2)$, что влечет
   $\max \check{T}(e_1,e_1') \le \min \check{T}(e_2,e_2')$. □

Далее нам понадобится лемма, связывающая отношение перехода на временных конфигурациях в исходной и детализированной временных структурах конфигураций.

Лемма 3.2. Пусть $TS \in \mathcal{TS}$, ref — функция детализации, $\check{T}\check{C},\check{T}\check{C}_1 \in TConf_{ref(TS)}$ и $\check{T}\check{C} \to \check{T}\check{C}_1$. Тогда существуют $TC, TC_1 \in TConf_{TS}$ такие, что $\check{T}\check{C}$ детализирует TC, $\check{T}\check{C}_1$ детализирует TC_1, и $TC \to TC_1$.

Доказательство следует из определения детализации временной конфигурации и отношения перехода. □

4. ПОВЕДЕНЧЕСКИЕ ЭКВИВАЛЕНТНОСТИ ВРЕМЕННЫХ СТРУКТУР КОНФИГУРАЦИЙ

В данном разделе мы вводим временное расширение трассовой эквивалентности и сохраняющей историю бисимуляции структур конфигураций из [9] в семантике частичного порядка, инвариантных относительно операции детализации структур конфигураций.

Сначала рассмотрим понятие временного частично упорядоченного множества.

Определение 4.1. *(Помеченным) временным частично упорядоченным множеством* (*над Act*) называется набор $TP = (E,\le,l,D)$, где

- $E \in \mathcal{P}_{\mathcal{F}}(\mathbf{E})$ — конечное множество событий;
- $\le\, \subseteq E \times E$ — частичный порядок;
- $l : E \to Act$ — помечающая функция, сопоставляющая событиям действия из *Act;*

- $D: E \to Interv$ — помечающая функция, сопоставляющая событиям временные интервалы.

Множество временных частично упорядоченных множеств обозначим через $\mathcal{TP}$.

Временные частично упорядоченные множества $TP_1 = (E_1, \leq_1, l_1, D_1)$, $TP_2 = (E_2, \leq_2, l_2, D_2)$ *изоморфны*, $TP_1 \cong TP_2$, если существует биективное отображение (*изоморфизм*) $f\colon E_1 \to E_2$, сохраняющее частичный порядок, помечающую и временную функции: $e <_1 e' \Leftrightarrow f(e) <_2 f(e')$, $l_1(e) = l_2(f(e))$, $D_1(e) = D_2(f(e))$ для всех $e, e' \in E_1$.

Для временной структуры конфигураций TS и $TC \in TConf_{TS}$ определим функцию $Tposet(TC) = (C, \leq_C, l_{TS}|_C, D_{TS}|_C)$, представляющую временные конфигурации в виде временных частично упорядоченных множеств.

Далее мы вводим понятия трассовой эквивалентности и сохраняющей историю бисимуляции временных структур конфигураций.

Определение 4.2.

- Назовем *языком* $TS \in \mathcal{TS}$ множество $L(TS) = \{ TP \in \mathcal{TP} \mid TP \cong Tposet(TC)$ для некоторой $TC \in TConf_{TS}\}$.
- Временные структуры конфигураций TS и TS' *трассово эквивалентны,* $TS \approx_{trace} TS'$, если $L(TS) = L(TS')$.

Пример 4.1. Язык временной структуры конфигураций TS из примера 3.1:

$$L(TS) = \{ \varnothing,\ a^{[d_1,d_1']},\ a^{[d_1,d_1']} \to b^{[d_2,d_2']} \mid [d_1,d_1'] \subseteq [0,2],\ [d_2,d_2'] \subseteq [d_1', \infty] \cap [1,3]\}.$$

Здесь и далее, непосредственная причинная зависимость между событиями, выражаемая частичным порядком, размечается стрелками, а события помечаются соответствующими действиями из Act.

Рассмотрим понятие сохраняющей историю бисимуляции временных структур конфигураций. Неформально говоря, две системы считаются бисимуляционно эквивалентны, если внешний наблюдатель не может различить их поведения.

Определение 4.3. Пусть TS и TS' — временные структуры конфигураций.

- Отношение $\mathcal{B}$, состоящее из троек (TC, f, TC'), где $TC \in TConf_{TS}$, $TC' \in TConf_{TS'}$, и $f\colon Tposet(TC) \to Tposet(TC')$ — изоморфизм, называется *hp-бисимуляцией*, если $((\varnothing,\varnothing),\varnothing,(\varnothing,\varnothing)) \in \mathcal{B}$ и для всех $(TC, f, TC') \in \mathcal{B}$ верно:

a) если $TC \to TC_1$ в TS, то $TC' \to TC'_1$ в TS' и $(TC_1, f_1, TC'_1) \in \mathcal{B}$, где $f \subseteq f_1$ для некоторых TC'_1 и f_1;
b) если $TC' \to TC_1'$ в TS', то $TC \to TC_1$ в TS и $(TC_1, f_1, TC'_1) \in \mathcal{B}$, где $f \subseteq f_1$ для некоторых TC_1 и f_1;
c) если $(TC, f\ TC') \in \mathcal{B}$, то $C \in \surd_{TS} \Leftrightarrow C' \in \surd_{TS'}$.

- TS и TS' *hp-бисимуляционно* эквивалентны, $TS \approx_{hpbis} TS'$, если между ними существует *hp*-бисимуляция.

Как и для безвременной модели, сохраняющая историю бисимуляция сильнее трассовой эквивалентности.

Теорема 4.1. Пусть TS и TS' — временные структуры конфигураций, тогда $TS \approx_{trace} TS' \Leftarrow TS \approx_{hpbis} TS'$.
Доказательство следует из определений эквивалентностей. □

Отметим, что обратная импликация неверна, что подтверждает следующий пример.

Пример 4.2. Рассмотрим временную структуру конфигураций $TS = (S,D)$ из примера 3.1 и $TS' = (S',D')$, где $Conf_{S'} = Conf_S \cup \{e_3\}$, $\surd_{S'} = \surd_S \cup \{e_3\}$, $l(e_3) = a$ и $D(e_3) = [2,2]$. Мы получаем $TS \approx_{trace} TS'$, но $\neg(TS \approx_{hpbis} TS')$, потому что после выполнения $a^{[2,2]}$ в TS всегда возможно выполнение $a^{[2,2]} \to b^{[2,3]}$ в отличие от TS'.

Далее исследуется влияние операции детализации действий временных структур конфигураций на рассмотренные выше эквивалентности. Для этого потребуется следующая вспомогательная лемма.

Лемма 4.1. Пусть $TS, TS' \in \mathcal{TS}$, ref — функция детализации, $TC \in TConf_{TS}$, $TC' \in TConf_{TS'}$, и $g: Tposet(TC) \to Tposet(TC')$ — изоморфизм. Тогда для любой $\check{T}\check{C}$, детализации TC, существует $\check{T}\check{C}'$, детализация TC', такая, что $\breve{g}: Tposet(\check{T}\check{C}) \to Tposet(\check{T}\check{C}')$ — изоморфизм, где $\breve{g}(e,e') = (g(e),e')$ для всех $(e,e') \in \check{C}$.
Доказательство следует по определению изоморфизма и детализации. □

Согласно следующей теореме, трассовая эквивалентность временных структур конфигураций инвариантна относительно операции детализации действий, что расширяет аналогичные результаты для моделей без встроенного времени [7, 9].

Теорема 4.2. Пусть TS и TS' — временные структуры конфигураций и ref — функция детализации, тогда

$$TS \approx_{trace} TS' \;\Rightarrow\; ref(TS) \approx_{trace} ref(TS').$$

Доказательство следует из леммы 4.1. □

В отличие от трассовой эквивалентности, сохраняющая историю бисимуляция временных структур конфигураций в общем случае не инвариантна относительно операции детализации действий, что подтверждает следующий пример.

Пример 4.2. Рассмотрим TS_1 и TS_2 из $\mathcal{TS}$, где $Conf_{TS_1} = \{\emptyset,\{e_1\}\}$, $Conf_{TS_2} = \{\emptyset,\{e_1\},\{e_2\}\}$, $\surd_{TS_1} = \surd_{TS_2} = \emptyset$, $l_{TS_1}(e_1) = l_{TS_2}(e_1) = l_{TS_2}(e_2) = a$, $D_{TS_1}(e_1) = D_{TS_2}(e_1) = [0,2]$ и $D_{TS_2}(e_2) = [0,1]$. Очевидно, что $TS_1 \approx_{hpbis} TS_2$. Но для функции $ref(a) = S$ из примера 2.1 мы получаем $\neg(TS_1 \approx_{hpbis} TS_2)$, так как после выполнения $a^{[0,1]}$ в TS_1 всегда возможно выполнение $a^{[0,1]} \rightarrow b^{[1,2]}$, а после выполнения $a^{[0,1]}$, помечающего e_2 в TS_2, возможно выполнение только $a^{[0,1]} \rightarrow b^{[1,1]}$.

В контексте моделей без встроенного времени [7, 9], сохраняющая историю бисимуляция инвариантна относительно операции детализации действий. Далее предлагается дополнительное условие для инвариантности данной эквивалентности в рамках временных структур событий.

Теорема 4.3. Пусть TS и TS' — временные структуры конфигураций, $TS \approx_{hpbis} TS'$, и ref — функция детализации. Тогда если существует hp-бисимуляция $\mathcal{B}$ между TS и TS' такая, что $\forall (TC,f,TC') \in \mathcal{B} \diamond D_{TS}|_C = D_{TS'}|_{C'}$, то

$$ref(TS) \approx_{hpbis} ref(TS').$$

Схема доказательства

Предположим, что $TS \approx_{hpbis} TS'$ и существует hp-бисимуляция $\mathcal{B}$ между TS и TS' такая, что $\forall (TC,f,TC') \in \mathcal{B} \diamond D_{TS}|_C = D_{TS'}|_{C'}$. Тогда, используя лемму 3.1, нетрудно показать, что отношение

$$\mathcal{R} := \{((C,T_1), g, (C',T'_1)) \mid ((C,T), g, (C',T')) \in \mathcal{B}, (C,T_1) \in Conf_{TS} \text{ и } T_1 = T'_1 \circ g\}$$

тоже является hp-бисимуляцией между TS и TS'. Далее, для $ref(TS)$ и $ref(TS)$ построим отношение $\check{\mathcal{R}} := \{ (\check{T}\check{C}, \check{g}, \check{T}\check{C}') \mid (TC, g, TC') \in \mathcal{R}$ и

- $\check{T}\check{C}$ — детализация TC с помощью ref, где $\check{C} = \bigcup_{e \in C} \{e\} \times C_e$ и $\check{T}(e,e') = T_e(e')$ для всех $(e,e') \in \check{C}$,

$TC_e = (T_e, C_e) \in TConf_{TS_e} \setminus \{(\varnothing,\varnothing)\}$,
$TS_e = (ref(l_{TS}(e)), D_e) \in \mathcal{TS}$ и
$D_e(e') = T(e)$ для всех $e' \in E_{TS_e}$;

– $\check{T}\check{C}'$ — детализация TC' с помощью *ref*, где $\check{C}' = \bigcup_{e \in C} \{g(e)\} \times C_e$ и $\check{T}'(g(e),e') = T_e(e')$ для всех $(e,e') \in \check{C}$;
– $\check{g}(e,e') := (g(e), e'))$ для всех $(e,e') \in \check{C}$. }

Проверим, что $\check{\mathcal{R}}$ является *hp*-бисимуляцией между *ref*(*TS*) и *ref*(*TS*).

1. Если $(\check{T}\check{C}, \check{g}, \check{T}\check{C}') \in \check{\mathcal{R}}$, то $\check{g} : Tposet(\check{T}\check{C}) \to Tposet(\check{T}\check{C}')$ — изоморфизм по лемме 4.1.
2. $((\varnothing,\varnothing),\varnothing,(\varnothing,\varnothing)) \in \check{\mathcal{R}}$, так как $((\varnothing,\varnothing),\varnothing,(\varnothing,\varnothing)) \in \mathcal{R}$.
3. Пусть $(\check{T}\check{C}, \check{g}, \check{T}\check{C}') \in \check{\mathcal{R}}$ и пусть $(TC, g, TC') \in \mathcal{R}$ — соответствующие их построению. Тогда
   a) если $\check{T}\check{C} \to \check{T}\check{C}_1$ в *ref*(*TS*), то по лемме 3.2 существуют $TC_2, TC_1 \in TConf_{TS}$ такие, что $\check{T}\check{C}$ детализирует TC_2, $\check{T}\check{C}_1$ детализирует TC_1, и $TC_2 \to TC_1$. Очевидно, что $TC_2 = (C,T_2)$ для некоторой $T_2 : C \to Interv$. По построению $\mathcal{R}$, получаем $(TC_2, g, TC_2') \in \mathcal{R}$ для $T_2' = T_2 \circ g^{-1}$. Легко показать, что $\check{T}\check{C}'$ детализирует TC_2'. Так как $\mathcal{R}$ является *hp*-бисимуляцией между *TS* и *TS'*, то существуют TC'_1 и g_1 такие, что $TC_2' \to TC'_1$ в *TS'*, $(TC_1, g_1, TC'_1) \in \mathcal{R}$ и $g \subseteq g_1$. Согласно построению $\check{\mathcal{R}}$, получаем $(\check{T}\check{C}_1, \check{g}_1, \check{T}\check{C}_1') \in \check{\mathcal{R}}$, где $\check{T}\check{C}_1'$ детализирует TC_1' и $\check{g} \subseteq \check{g}_1$, что влечет $\check{T}\check{C}' \to \check{T}\check{C}_1'$ в *ref*(*TS'*);
   b) симметрично пункту (a);
   c) по построению $\check{\mathcal{R}}$ и определению $\surd_{ref(TS)}$ получаем
   $\check{C} \in \surd_{ref(TS)} \Leftrightarrow \check{C}' \in \surd_{ref(TS')}$. □

5. ЗАКЛЮЧЕНИЕ

В работе исследовалась операция детализации действий структур конфигураций с глобальным непрерывным временем и вопрос инвариантности трассовой эквивалентности и сохраняющей историю бисимуляции в семантике временных частичных порядков. Было показано, что трассовая эквивалентность, в отличие от сохраняющей историю бисимуляции, инвариантна относительно операции детализации временных структур конфигураций. Для сохраняющей историю бисимуляции было дано дополнительное условие ее инвариантности.

В дальнейшем планируется дополнить полученные результаты, а также расширить их на другие классы временных структур конфигураций и временных структур событий.

СПИСОК ЛИТЕРАТУРЫ

1. Aceto L. History preserving, causal and mixed-ordering equivalence over stable event structures // Fundamenta Informaticae. — 1992. — Vol. 17, N 4. — P. 319–331.
2. Alur R., Dill D. The theory of timed automata // Theor. Comput. Sci. — 1994. — Vol. 126. — P. 183–235.
3. Alur R., Henzinger T.A. Logics and models of real time: a survey // Lect. Notes Comput. Sci. — 1992. — Vol. 600. — P. 74–106.
4. Andreeva M. V., Virbitskaite I. B. Observational Equivalences for Timed Stable Event Structures // Fundamenta Informaticae. — 2006. — Vol. 72. — P. 1–19.
5. Čerāns. K. Decidability of bisimulation equivalences for parallel timer processes // Lect. Notes Comput. Sci. — 1993. — Vol. 663. — P. 302–315.
6. Darondeau Ph., Degano P. Refinement of actions in event structures and causal trees // Theore. Comput. Sci. — 1993. — Vol. 118. — P. 21–48.
7. Van Glabbeek R.J., Goltz U. Equivalence notions for concurrent systems and refinement of actions // Lect. Notes Comput. Sci. — 1989. — Vol. 379. — P. 237–248.
8. Van Glabbeek R.J., Goltz U. Refinement of actions in causality based models // Lect. Notes Comput. Sci. — 1990. — Vol. 430. — P. 267-300.
9. Van Glabbeek R.J., Goltz U. Refinement of actions and equivalence notions for concurrent systems // Acta Informatica. — 2001. — Vol. 37. — P. 229–327.
10. Van Glabbeek R.J., Plotkin G. D. Configurations structures (extended abstract) // Proc. 10th Annual IEEE Simp. on Logic in Computer Science (LICS'95), San Diego, USA, 1995. — IEEE Computer Society Press, 1995. — P. 199–209.
11. Hoare C.A.R. Communicating sequential processes. — Prentice-Hall, London, 1985.
12. Murphy D. Time and duration in noninterleaving concurrency // Fundamenta Informaticae. — 1993. — Vol. 19. — P. 403–416.
13. Steffen B., Weise C. Deciding testing equivalence for real-time processes with dense time // Lect. Notes Comput. Sci. — 1993. — Vol. 711. — P. 703–713.
14. Vaandrager F.W. An explicit representation of equivalence classes of the history preserving bisimulation. — Manuscript, CWI-Amsterdam, 1989.
15. Virbitskaite. I.B. An observation semantics for timed event structures // Lect. Notes Comput. Sci. — 2001. — Vol. 2244. — P. 215–225.
16. Winskel G. Event structures. In Edvances in Petri Nets: Part II // Lect. Notes Comput. Sci. — 1987. — Vol. 255. — P. 325–392.
17. Winskel G. An introduction to event structures // Lect. Notes Comput. Sci. — 1988. — Vol. 354. — P. 364–397.
18. Weise C., Lenzkes D. Efficient scaling-invariant checking of timed bisimulation // Lect. Notes Comput. Sci. — 1997 — Vol. 1200. — P. 176–188.

Я. Н. Батура

ЧЕЛОВЕКО-МАШИННАЯ МОДЕЛЬ ЯЗЫКА МЫШЛЕНИЯ

1. ВВЕДЕНИЕ

Традиционный подход к представлению знаний в искусственном интеллекте основывается на символьной логике [1]. Такой подход, с одной стороны, подробно разработан и обоснован теоретически, а с другой стороны — наиболее удобно реализуем и позволяет легко оценивать получаемые результаты: «символьная» сторона мышления тесно связана с языком, который служит основным средством передачи знаний между мыслящими индивидуумами. Вместе с тем, «мощность» символьно-логического подхода заведомо меньше «мощности» человеческого разума, деятельность которого включает символьную логику, но не заключается в ней. Подтверждением этому, например, служит невозможность добиться стопроцентной точности в задаче анализа текстов на естественном языке: различные неязыковые факторы, помогающие человеку делать правильный выбор в понимании смысла текста, недоступны машине, что и снижает качество ее деятельности. Многочисленные попытки навести мосты между символьной логикой и естественными языками не увенчались успехом по той же самой причине. Более молодой (хотя уже и насчитывающий более тридцати лет), несимвольный, подход к представлению знаний ориентируется на непосредственное восприятие разумным существом окружающего мира через органы чувств, выделение схожих элементов в поступающих информационных потоках, классификацию этих элементов и т. д. К этому подходу относятся и нейронные сети.

Человеческий разум представляет собой единый комплекс, работающий и с символьной, и с несимвольной информацией. Поэтому модель искусственного разума, хотя бы отдаленно похожего на человеческий, должна обладать обеими этими способностями. Попытки совместить работу с символьной и несимвольной информацией предпринимались неоднократно. В настоящей работе предлагается абстрактная модель представления знаний («языка» мышления), которая может найти свои параллели в разуме человека и, в то же время, служить основой для конкретных машинных моделей.

2. МОДЕЛИ РАЗУМА

На рис. 1 приведена иерархия рассуждений о разуме. Более высокие уровни ближе к философии, а более низкие — к прикладным наукам. Переход с уровня на лежащий выше соответствует синтезу (абстрагированию), а на лежащий ниже — анализу (причем конструктивному). В нашем случае мы исходим из предположения, что «разумность вообще» (высокий уровень) определяется совокупностью определенных феноменов (средний уровень). Совокупность этих феноменов обеспечивается устройством конкретных моделей разума, которые мы рассматриваем. В свою очередь, в рассуждении о моделях разума можно также выделить два уровня, нижний из которых соответствует «реализации» (если этот термин применим к человеку как к разумному существу).

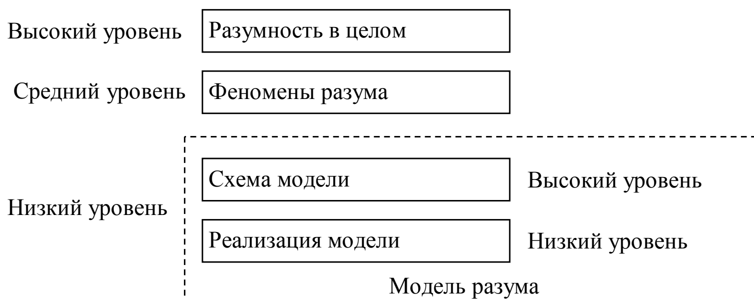


Рис. 1

Складывается впечатление, что подавляющее большинство работ в области искусственного интеллекта не выходят за рамки нижнего уровня (более того, реализуются не модели в целом, а отдельные элементы их), а работы в области философии (конкретнее, эпистемологии) — не рискуют опуститься ниже среднего уровня феноменов разума.

Вопрос о том, что же такое разум (в конструктивном смысле), еще не разрешен. Множество феноменов, необходимых и достаточных для разумности, не определено, но один из них, необходимый для разумности, всё же можно назвать (он практически очевиден): *наличие механизма внутреннего представления окружающего мира, позволяющего осуществлять внутреннее моделирование*.

Далее следует описание модели представления знаний, соответствующее уровню Схемы моделей из рис. 1. Эта модель близка к человеческой, и

в ней выделяется несимвольный и символьный уровни, а также связь между ними. Модель раскрывает и дополняет понятие “интраязыка” в [2].

3. НЕСИМВОЛЬНЫЙ УРОВЕНЬ

Несимвольный уровень представляет собой фундамент мышления, над которым надстраивается абстрактный, символьный уровень.

3.1. Сферы восприятия

Несимвольная информация напрямую связана с восприятием разумным существом окружающего мира посредством органов чувств $P_1,\ldots,P_N$. Можно считать поступающие от этих органов чувств потоки условно-независимыми: при «перекрытии» одного из них информация будет продолжать поступать от других (так, человек может закрыть глаза, но продолжать слышать звуки и пробираться на ощупь). Можно также считать, что обработка этих потоков информации на низком уровне также осуществляется независимо, и выделить *сферы восприятия* $S_1,\ldots,S_N$, соответствующие органам чувств. Каждая сфера восприятия, вообще говоря, может иметь собственный «формат» представления информации. У человека важнейшими сферами восприятия являются зрительная и слуховая.

3.2. Потоки восприятия

Разумное существо воспринимает мир не как набор статических картинок, а в динамике. Каждая сфера восприятия соответствует органу чувств, непрерывно потоком подающих информацию. Так, например, слуховая память человека может содержать огромное количество музыкальных мотивов, которые он может восстанавливать произвольно или по отрывкам. Музыка, которую помнит человек, — это звуковой образ, подчиняющийся определенному ритму. На любой момент времени существует некоторый объем информации, поданной потоком. Назовем этот объем *реальным срезом восприятия*. С течением времени часть информации, поступившая раньше, уходит из среза информации, и одновременно в срез поступает новая информация, соответствующая изменению окружающего мира. Совокупность реальных срезов информации, относящихся ко всем имеющимся сферам восприятия, назовем *реальной ситуацией*.

3.3. Абсолютные внутренние образы

Одним из базовых понятий нашей модели является *абсолютный внутренний образ* — информация, относящаяся к определенной сфере восприятия, которая может использоваться для проверки наличия в соответствующем срезе восприятия прообраза этого внутреннего образа. Будем обозначать абсолютные внутренние образы строчными латинскими буквами.

Функция проверки — важнейшая функция (механизм) сферы восприятия, будем обозначать ее $V_i(t)$. Она возвращает «истину» (или «срабатывает»), если в соответствующем срезе восприятия присутствует прообраз абсолютного внутреннего образа t, и «ложь» в противном случае. Абсолютный внутренний образ, по сути, является инвариантом для любых срезов восприятия, в которых фигурирует прообраз t. Отметим, что у человека аналог этого «инварианта» вполне соответствует названию (по инварианту, вообще говоря, бывает невозможно восстановить исходный объект). У различных сфер восприятия человека эта «невозможность» проявляется в разной степени. Как уже упоминалось выше, в слуховой сфере инвариант довольно близок к прообразу. Музыкальное произведение, которое помнит человек, может отличаться от исходного услышанного по тональности или по темпу. В зрительной же сфере инвариант весьма сжат. Интересно отметить, что для разных элементов действительности, воспринимаемых человеком с помощью зрения, «точность» этих инвариантов значительно отличается друг от друга. Максимальная точность инвариантов, по всей видимости, имеется у внутренних образов лиц людей, соответствующих расе, среди представителей которой проживает (или общается) данный человек. Так, для русского человека «все китайцы на одно лицо», а для китайца «на одно лицо» все западные люди. Лица же людей родной расы обладают для человека максимальной индивидуальностью. Но несмотря на то что узнавать и отличать лица людей легко для каждого человека, восстановить по памяти лицо человека под силу только очень опытным художникам, да и то процесс восстановления проходит по принципу проб и ошибок.

С функцией проверки тесно связан процесс распознавания внутренних образов в срезе восприятия, проходящий в реальном времени. Некий механизм сопоставляет потоку множество внутренних образов таких, что на них «выполняется» функция проверки на текущем срезе восприятия.

3.4. Относительные внутренние образы

Внутренние образы — отражение окружающей действительности. Большая часть окружающих элементов действительности имеет сложную природу, и в них можно выделить подэлементы. Сфера восприятия разумного существа должна уметь «работать» с составными элементами окружающей действительности, следовательно, должен существовать механизм, отражающий во внутренних образах структуру их прообразов.

Относительный внутренний образ — это внутренний образ («роль»), связанный с другим, абсолютным или относительным, внутренним образом («хозяином»). Относительный внутренний образ определяет некоторую роль (или подэлемент) в образе-хозяине. Будем обозначать относительный внутренний образ парой s/t, где s — это роль, а t — это хозяин. С относительными внутренними образами сферы восприятия S_i свяжем соответствующую *функцию относительной проверки* $R_i(x, s/t)$, которая возвращает «истину», если в потоке восприятия внутренний образ x играет роль s во внутреннем образе t.

С функцией относительной проверки тесно связан процесс распознавания ролей в срезе восприятия, связывающий уже распознанные внутренние образы относительными внутренними образами, т. е. отношениями «роль — хозяин». Поясним это примером.

Обозначим зрительную сферу восприятия номером 1 (т. е. S_1). Рассмотрим внешний образ (рисунок или фотографию) женщины, держащей букет цветов. В нем могут быть распознаны три абсолютных и два относительных внутренних образа:

s_1	Женщина
s_2	Букет цветов
s_3	Держать

t_1/s_3	Тот, кто держит
t_2/s_3	То, что держат

В результате в срезе восприятия будут выполняться функции $V_1(s_1)$, $V_1(s_2)$, $V_1(s_3)$, $R_1(s_1, t_1/s_3)$, $R_1(s_2, t_2/s_3)$.

Другой пример. Ребенок знает понятие «человек» и «рука» и может показать, где у человека рука. С другой стороны, он может самостоятельно определить как «руку» воткнутую в снеговика ветку. Относительный внутренний образ в данном случае — информация, позволяющая распознавать «руки» у объектов.

3.5. Образы и временные последовательности

Так как в общем случае внутренние образы представляют собой след восприятия среза потока, то они захватывает изменение своих прообразов во времени. Аналогично механизму увязывания образов по ролям, в нашей модели несколько важнейших механизмов будут отвечать за выделение статичных или более коротких по длительности подобразов из основного образа, сравнение нескольких подобразов на предмет следования друг за другом (или одновременности) по времени, проверку присутствия внутреннего образа как подобраза по времени в другом образе и т. д. Эта функциональность будет лежать в основе внутреннего моделирования, осуществляемого субъектом в нашей модели.

3.6. Внутреннее представление срезов восприятия

Множество абсолютных образов и связывающих их относительных образов (ролей), распознанных в срезе восприятия, назовем его *внутренним представлением*.

В принципе, начиная с момента, когда в срезе восприятия распознан внутренний образ, возникает и первый символ, который может уже использоваться в различных выражениях. Но такой символ ограничивался бы лишь только своей сферой восприятия. Человеческий же разум оперирует с символами, одновременно увязанными с множеством различных сфер восприятия.

4. СИМВОЛЬНЫЙ УРОВЕНЬ

Настоящий символ возникает тогда, когда появляется ассоциация внутренних образов из различных сфер восприятия. Так, внешность знакомого человека ассоциируется с его голосом и его именем (не будем пока уточнять, к какой сфере восприятия относится имя, т. е. слово языка), вид яблока ассоциируется с его запахом и вкусом, и т. д.

Внутренним символом (будем обозначать их строчными греческими буквами) является непустое множество из одного или нескольких внутренних образов (не обязательно из различных сфер восприятия) и/или нескольких других внутренних символов, на которое можно ссылаться как на единое целое. На уровне реализации внутренний символ может быть представлен узлом, связывающим набор ссылок на внутренние образы или символы.

Связь внутреннего символа α с образующими $t_1,\ldots,t_k,\beta_1,\ldots,\beta_l$ будем обозначать $\alpha[t_1,\ldots,t_k,\beta_1,\ldots,\beta_l]$, а множество образующих символа α — как $G(\alpha)$.

4.2. Связи между внутренними символами

Можно выделить три основных вида связи между внутренними символами.

1. *Ролевая связь* («$\alpha \underset{s/t}{-} \beta$») — это связь между двумя внутренними символами α и β такая, что существуют абсолютные внутренние образы $s_m \in G(\alpha)$ и $t \in G(\beta)$ и относительный внутренний образ s/t, относящиеся к одной сфере восприятия S_i, такие, что в некотором срезе восприятия, соответствующем этой сфере, выполняется $R_i(s_m, s/t)$.

2. *Временнáя связь* ($\alpha \to \beta$). Связь между двумя базовыми символами в некоторой ситуации $\alpha[\ldots s \ \ldots]$ и $\beta[\ldots t \ldots]$, существующая при условии, что s и t из одной сферы восприятия и s в этой ситуации происходит раньше, чем t.

3. *Ассоциация* (обозначается $\alpha \cdots \beta$). Связь между двумя внутренними символами с общим образующим внутренним образом.

Внутренним выражением будем называть множество связанных между собой внутренних символов.

4.3. Связь языка с внутренними символами

Что значит «понимать язык»? Лучше всего проанализировать это явление в его естественном развитии.

Обычно овладение русским языком начинается с формирования ассоциаций между предметом и соответствующим ему именем существительным, которое называют ребенку. Очевидно, эта сфера восприятия — слуховая, но для удобства будем называть ее речевой. Повторение слова формирует у ребенка внутренний образ w этого слова, а упоминание вместе с каким-либо предметом (внутренний образ которого обозначим o) формирует внутренний символ $\alpha[w,o]$. В результате в потоке речи ребенок начинает распознавать символы, образующие слова которых он уже знает. Вся остальная речь пропускается им как не содержащая ничего ценного.

Следующий этап в овладении языком — формирование относительных внутренних образов в речевом потоке восприятия. Простейшие предложения — «машинка упала», «мама пришла» и т. д. — соответствуют ситуации, воспринимаемой уже с двумя абсолютными и одним относительным внутренними образами. Так, ситуации из предложения «машинка упала» будет соответствовать символьное описание $\alpha \underset{s/t}{-} \beta$, где $\alpha[t_1,t_2]$ — это внутренний символ, соответствующий «машинке», образованный внутренним зрительным образом t_1 «машинки» и внутренним образом t_2 слова «машинка», $\beta[t,t_3]$ — внутренний символ, соответствующий падению, с образующими t — внутренним зрительным образом падения и t_3 — внутренним образом глагола «падать». При порождении символьного описания играет роль относительный внутренний образ «то, что падает» s/t, связывающийся при обучении с соответствующим внутренним относительным речевым образом («Х падаю/падаешь/падает/упадешь» и т.д., делающим акцент на Х — «исполнителе» падения). Эта связь закрепляется путем повторения этих слов в различных сочетаниях, и у ребенка формируются относительные внутренние образы речевой сферы, соответствующие одновременно и грамматическим конструкциям русского языка, и ролям в ситуациях (вначале — простейших ситуаций). В результате постоянного общения со взрослыми у ребенка формируется так много внутренних относительных речевых образов, что даже из одного предложения «машинка упала» выделяется масса информации — о роде подлежащего, прошедшем времени и т. д. (все это относительные внутренние образы).

Третий этап — формирование внутренних символов, образуемых на основе связи образов слов с целыми внутренними выражениями. Эти внутренние символы приходят, главным образом, из языка и связаны с абстрактными понятиями («хороший», «плохой»).

5. МЫШЛЕНИЕ

В общем случае мышление в нашей модели есть процесс возникновения и преобразования внутренних выражений относительно определенного контекста. Контекстом мышления может быть как отражение реальной ситуации, так и воображаемая внутренняя модель. Сами по себе внутренние выражения, возникающие и изменяющиеся в процессе мышления, имеют то же внутреннее устройство, что и внутренние выражения, хранящиеся в дол-

говременной памяти субъекта, и на уровне реализации представляют собой взаимоувязанные ссылки на имеющиеся внутренние образы и символы.

В процессе мышления активно используются следующие механизмы.

1. Выделение ролей или хозяев. К некоторому символу α выражения добавляется внутренний символ β, если он может быть подчинен α некоторой ролевой связью в данном контексте: $\beta \underset{s/t}{-} \alpha$, и наоборот.

2. Замена по ассоциации. Если в долговременной памяти имеется ассоциация $\alpha \cdots \beta$, то символ β в выражении может быть заменен на α, и наоборот. Полученное в результате выражение может быть неадекватным.

3. Добавление по ассоциации. Если в долговременной памяти имеется ассоциация $\alpha \cdots \beta$, то к символу α может быть ассоциативно присоединен внутренний символ β.

4. Удаление символов в выражении.

5. Шаг моделирования по времени. Если одна или несколько образующих символов, имеющихся в выражении, представляет собой временну́ю последовательность, то на основе выделения подобразов из этой образующей может происходить добавление к выражению внутренних символов, образованных этими подобразами.

6. Проверка адекватности. Имеющееся выражение сопоставляется информации, имеющейся в контексте рассуждений (реальной ситуации или условиям модели).

СПИСОК ЛИТЕРАТУРЫ

1. Luger, G.F. Artificial Intelligence: Structures and Strategies for Complex Problem Solving. — London: Addison-Wesley, 2002.
2. Батура Я.Н. Подход к моделированию самообучающихся субъектов // Тр. VII национальной конф. по искусственному интеллекту с международным участием (КИИ'2002). — Коломна, 2002.

Д. М. Белоглазов

ОБНАРУЖЕНИЕ ВЗАИМОДЕЙСТВИЯ ФУНКЦИОНАЛЬНОСТЕЙ В ТЕЛЕФОННЫХ СЕТЯХ С ПОМОЩЬЮ РАСКРАШЕННЫХ СЕТЕЙ ПЕТРИ

1. ВВЕДЕНИЕ

Проблема взаимодействия функциональностей (feature interaction problem, FIP) [2] уже в течение нескольких десятков лет изучается мировым научным сообществом, так как является одной из наиболее интересных, практически значимых и сложных проблем в области информационных технологий.

Проблема взаимодействия функциональностей заключается в том, что в комплексных системах, включающих в себя несколько модулей со смежной функциональностью, модули могут взаимодействовать друг с другом, вызывая тем самым отклонения от ожидаемого поведения системы в целом. Такие отклонения могут быть безвредными, однако во многих случаях они оказываются нежелательными и даже опасными.

Классический пример такой системы — телефонные сети с дополнительными функциональностями (сервисами). В нашей работе FIP-проблема рассматривается именно в этом контексте. Для выявления взаимодействия функциональностей телефонных сетей [2] используется метод проверки моделей (model checking method) [6] для графов достижимости модифицированных раскрашенных сетей Петри [5, 8], моделирующих телефонные сети, где проверяемые свойства выражены формулами мю-исчисления.

2. МЕТОДЫ И СРЕДСТВА ВЕРИФИКАЦИИ

В качестве моделей в данной работе используются иерархические временные типизированные сети (ИВТ-сети) [5, 8] — модификация классических раскрашенных сетей Петри (РСП) [1]. Напомним некоторые отличия ИВТ-сетей от классических РСП.

Типы, используемые в ИВТ-сетях, строятся на основе стандартных: целого, вещественного, строкового и булевского. Множество цветов задаётся

перечислением всех возможных значений. Составные типы (массивы и записи) представляются в виде кортежей.

Для верификации изучаемых систем использовался программный комплекс SPV (SDL Petri Net Verifier) [5, 8], разработанный в лаборатории теоретического программирования Института систем информатики СО РАН. Общая схема верификации при помощи системы SPV такова: SDL-спецификация транслируется в РСП (двух видов — классические РСП Йенсена и ИВТ-сети), для которых затем строятся графы достижимости, являющиеся конечными моделями.

В данной работе использовалась только часть модулей системы SPV, так как основную часть верифицируемых моделей составляли ИВТ-сети, построенные вручную. Эта часть системы имеет своё собственное название — Petri Net Verifier (PNV) [3, 5, 7]. Схема работы с компонентом PNV такова: для ИВТ-сети строится граф достижимости с полным описанием вершин. Далее, на основании описания вершин и описания исследуемых свойств (предикатов) в терминах ИВТ-сети при помощи блока построения предикатов определяются вершины графа, в которых эти предикаты истинны. Блок проверки моделей на вход получает три файла: формулу мю-исчисления, описывающую исследуемое свойство, описание предикатов, используемых в формуле, и граф достижимости, на котором проверяется это свойство. На выходе этого блока мы имеем набор состояний, в которых истинно проверяемое свойство, либо ALL STATES — в случае, если формула истинна во всей модели, либо false, если формула ложна.

3. БАЗОВАЯ МОДЕЛЬ ЗВОНКОВ

В основе любой телефонной сети лежит так называемая «базовая модель звонков» (basic call state model, BCSM) [2, 4]. Её же называют Plain Old Telephone Service или POTS. Это модель простейшего телефонного сервиса, которая позволяет абонентам общаться — набирать номер, отвечать на звонки. Примером правила, описываемого в рамках BCSM, является такое: «Абонент, снявший трубку, слышит длинный гудок». Или, например, «абонент, набравший занятый номер, слышит короткие гудки».

Согласно набору таких правил (приведённых, например, в [4]) построена ИВТ-сеть, моделирующая работу BCSM. Количество станций является параметром сети, и может быть изменено. Модель для BCSM приведена на рис.1. На рисунке опущены охранные функции на переходах, а также некоторые вспомогательные места и переходы.

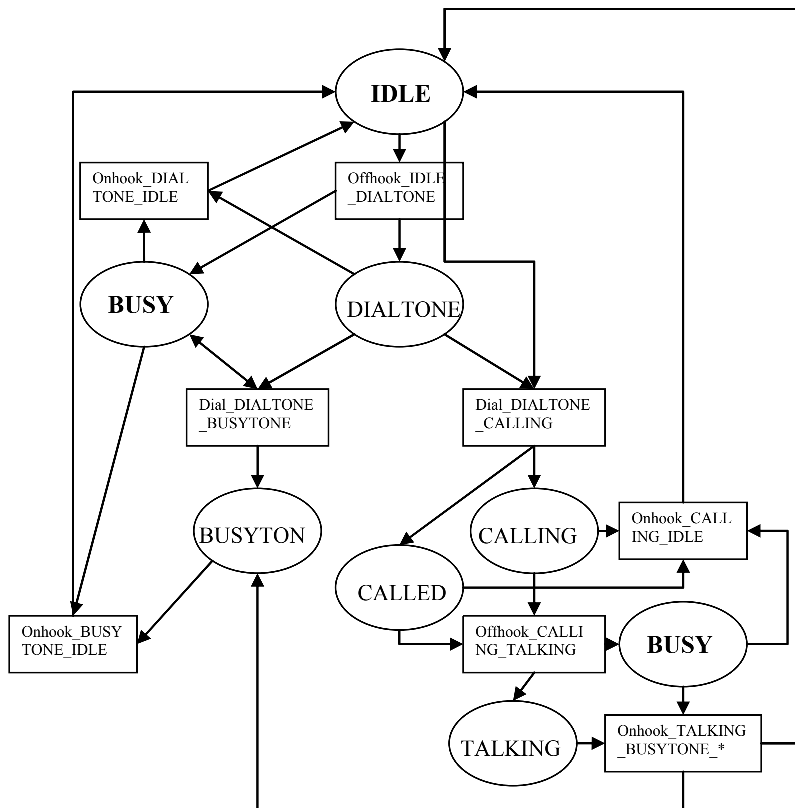


Рис.1. ИВТ-сеть для Basic Call State Model

Поясним строение сети на рис.1. Места сети обозначают состояния абонентов:

- **IDLE** — абонент бездействует, трубка телефона повешена,
- **DIALTONE** — абонент слышит непрерывный гудок,
- **BUSYTONE** — абонент слышит короткие гудки,
- **BUSY** — абонент занят (не бездействует), трубка снята,
- **CALLING** — абонент звонит другому абоненту,
- **CALLED** — абоненту звонит другой абонент,
- **TALKING** — пары общающихся абонентов.

Все места, кроме TALKING, имеют целочисленный тип. Место TALKING имеет тип записи — пары двух целочисленных.

Все переходы в сети делятся на три группы (по первой части названия):

- **Offhook** — абонент снимает трубку,
- **Dial** — абонент набирает номер другого абонента,
- **Onhook** — абонент кладёт трубку.

Поясним на простом примере.

Допустим, изначально в системе 3 абонента. Они находятся в состоянии IDLE.

Это означает, что в месте IDLE находятся три фишки со значениями 1, 2 и 3.

IDLE : 1 2 3

Далее, допустим, абонент 1 снимает трубку. По переходу Offhook_IDLE_DIALTONE он попадает в состояние BUSY и слышит непрерывный гудок. Разметка:

IDLE : 2 3
BUSY : 1
DIALTONE : 1

Предположим, что 1 набирает номер 2. Следуем переходу Dial_DIALTONE_Calling, получаем разметку:

IDLE : 2 3
BUSY : 1
CALLING : 1
CALLED : 2

Абонент 2 снимает трубку (Offhook_CALLING_TALKING):

IDLE : 3
BUSY : 1 2
TALKING : [1,2]

Оставляем читателю возможность проследить дальнейшие варианты развития событий в примере.

4. ИССЛЕДУЕМЫЕ ФУНКЦИОНАЛЬНОСТИ

В построенную модель для BCSM были внедрены следующие функциональности.

- **CW (Call Waiting)**

Если абонент *y* звонит абоненту *x*, а *x* занят и подключён к CW, то *y* отображается на дисплее *x*, и *x* может ответить на звонок *y*.

- **Call Forwarding when Busy (CFB)**

Если абонент *y* звонит абоненту *x*, а *x* занят и подключён к CFB с перенаправлением на номер абонента *z*, то *y* соединяется с абонентом *z*.

- **Emergency Call (EMG)**

Если абонент *y* звонит абоненту *x*, а *x* подключён к EMG, то в случае, когда *y* кладёт трубку и берёт её снова, связь восстанавливается. Разрывается связь только в том случае, когда трубку кладёт *x*.

- **Denied Termination (DT)**

Если абонент *x* подключён к DT, то все звонки на номер *x* запрещены. Абонент, набирающий номер *x*, автоматически получает сигнал «занято».

- **Direct Connect (DC)**

Если абонент *x* подключён к DC с номером *y*, то как только *x* снимает трубку, он автоматически соединяется с абонентом *y*.

Функциональности добавлялись путём внесения изменений в сетевую модель BCSM. Например, добавлялись охранные условия на некоторых переходах, вводились дополнительные переходы и места. Функциональности внедрены удобным образом, введено дополнительное место FEATURES типа запись, каждое поле которой имеет тип int. Структура этой записи такова:

- stationID — номер станции, для которой задаются функциональности;
- CW — 1 или 0, обозначает, подключена ли эта услуга (1 — да, 0 — нет);
- CFB — номер станции, на которую переводится звонок или 0, если услуга не подключена;
- EMG — 1 или 0;
- DT — 1 или 0;
- DC — номер вызываемой станции, либо 0, если услуга отключена;

Таким образом, для того чтобы добавить или удалить функциональность для абонента, достаточно поменять значения соответствующих полей в фишке, лежащей в месте FEATURES. Например, наличие в месте FEATURES фишки [1,1,2,0,1,0] означает, что абонент №1 подключил себе услуги CW, CFB и DT, причём, для CFB адресом перенаправления задан телефон абонента №2.

Для верификации использовались модели со всевозможными комбинациями этих функциональностей, количество абонентов варьировалось от 2 до 5.

5. ВЕРИФИКАЦИЯ МОДЕЛИ ЗВОНКОВ С ВЗАИМОДЕЙСТВУЮЩИМИ ФУНКЦИОНАЛЬНОСТЯМИ

Для построенных с помощью PNV моделей были проверены следующие свойства.

1. **Наличие тупиков.**
   Тупиком считается такое состояние системы, из которого переход в какое-либо другое состояние невозможен.
   Для спецификации этого свойства на модели используется формула ¬<to>*true,* где *true* — предикат тождественной истинности — все вершины графа.
2. **Зацикливание**,
   что означает наличие таких циклов, из которых невозможно вернуться в исходное состояние.
   Если имеется набор P подозрительных на зацикливание состояний, то наличие циклов проверяется μ-формулой P→μX.(<to>(X ∨ P)). Однако для моделей без тупиков ситуация упрощается: зацикливание совпадает с нарушением условия «возможность вернуться в исходное состояние», выражаемого формулой μX.(<to>(*begin* ∨ X)), где *begin* — предикат, истинный в начальном состоянии.
3. **Недетерминизм,**
   что означает наличие в ИВТ-сети конфликтующих переходов, соответствующих двум функциональностям.
   Недетерминизм для двух функциональностей проверяется следующим образом. Заводятся предикаты, соответствующие разметкам ИВТ-сети, в которых возможно срабатывание перехода для каждой функциональности. Обозначаем эти предикаты, например, по названиям исследуемых функциональностей с добавлением префикса en (от enabled). Заводим также предикаты для разметок, полученных в результате срабатывания функциональностей по названиям функциональностей. Для CW и CFB проверяемая формула будет иметь вид: (enCW*enCFB)&<to>(CW&¬(enCFB)). Формула истинна в моделях, у которых существует переход из разметки, в кото-

рой возможно срабатывание обеих функциональностей, в разметку, которая получена в результате срабатывания CW, и в которой невозможно срабатывание CFB. Это и есть недетерминизм.

4. **Нарушение условий в функциональностях**.
В данном случае условие есть только у Denied Termination, это условие — «никто никогда не может звонить абоненту, подключившему услугу DT». Нарушение этого условия просматривается на этапе построения графа достижимости. Если есть хоть одна вершина, соответствующая разметке, в которой TALKING содержит фишку [*y*,*x*], а *x* при этом подключил услугу DT (т. е. в месте FEATURES есть фишка вида [*x*,…,…,…,1,…]), то условие нарушено.
Результаты проведённых экспериментов представлены в таблице 1.

Таблица 1

Взаимодействие функциональностей

Функциональности	Тупики	Зацикливание	Недетерминизм	Нарушение условий
CW + CFB	false	false	**True**	false
CW + DC	false	false	False	false
CW + DT	false	false	**True**	**true**
CFB + DC	false	false	False	false
CFB + DT	false	false	**True**	**true**
DC + DT	false	false	false	**true**
EMG + EMG	false	**true**	false	false
CW+CFB+DT	false	false	**true**	**true**
CW+DC+DT	false	false	**true**	**true**
DC+EMG+EMG	false	**true**	false	false
DC+EMG+EMG*	false	**true**	**true**	false
All Features	false	**true**	**true**	**true**

Поясним некоторые результаты. Как видно, функциональности CW и CFB взаимодействуют, причём в их взаимодействии возникает недетерминизм, который в данном случае означает, что сработать может любой из

двух вариантов, при этом второй исключается. Действительно, предположим, что абонент подключил себе обе эти услуги. Подключение CW означает, что если номер абонента занят, все входящие звонки отображаются на дисплее и удерживаются, но, с другой стороны, подключение CFB означает в этом случае перенаправление всех входящих на третий номер. Возникает неоднозначность выбора, которая и отражена в результате эксперимента.

Другое интересное явление можно наблюдать в случае, когда два абонента подключают услугу EMG и один из них звонит другому. Стоит отметить, что данной услугой пользуются обычно такие организации как полиция, служба спасения и т.п. Представим, что полиция набирает номер службы спасения. Как итог получаем разговор, который ни одна из сторон не может прекратить по условиям EMG.

В экспериментах, в которых была подключена функциональность DT, можно отметить нарушение условий для DT во всех исследуемых сочетаниях, а также недетерминизм, возникающий при совмещении этой функциональности с CW и CFB. При подключении всех этих услуг вместе также возникает недетерминизм, причём в этом случае срабатывание одной функциональности блокирует срабатывание двух других.

Случай DC+EMG+EMG показателен в том смысле, что используемая в исследованиях модель накладывает свои ограничения на возможное взаимодействие. Представим себе такую ситуацию. Есть два абонента A и B, у которых подключена услуга EMG, причём у A также подключена услуга DC, связывающая его с третьим абонентом C. Допустим, B позвонил A. Теперь, если A кладёт трубку, возникает вопрос: в какое состояние переходит A? Если в BUSY, как это реализовано в нашей модели, остаётся только зацикливание, недетерминизма не возникает. А если в IDLE, то при снятии трубки абонентом A возникает недетерминизм, так как возможно возвращение в цикл EMG+EMG, но при этом возможно срабатывание DC, при котором A соединяется с C и выходит из зацикливания. Этот случай отмечен в Таблице 1 звёздочкой.

И наконец, при добавлении всех функциональностей, как и следовало ожидать, возникли зацикливания, недетерминизм и нарушение условия для DT.

Для исследования роста модели в зависимости от количества абонентов были построены графы достижимости для BCSM с количеством пользователей от 2 до 5. Результаты в таблице 2.

Таблица 2

Рост модели в зависимости от числа абонентов

#абонентов	2	3	4	5	6
#вершин графа	14	77	468	2967	20294

6. ЗАКЛЮЧЕНИЕ

Итак, в данной работе исследована важная проблема взаимодействия функциональностей телефонных сетей. Построена ИВТ-сеть, представляющая базовую модель звонков телефонной сети с пятью функциональностями. Проведена программная верификация взаимодействия этих функциональностей в данной сетевой модели.

Отметим, что комбинация методов, используемых в данной работе, не упоминается ни в одном из многочисленных изученных источников. Эти методы, использовавшиеся прежде для верификации протоколов кольцевых сетей, хорошо показали себя на новом классе задач — проблеме взаимодействия функциональностей в телефонных сетях. Нашим преимуществом является возможность проверки произвольного свойства, выраженного на языке μ-исчисления.

Для сравнения результатов нашей работы с результатами [4] были проведены эксперименты с аналогичными моделями для 3-х абонентов и некоторых комбинаций функциональностей (таблица 1). Основным способом задания моделей в работе [4] являются так называемые спецификации, заданные правилами (rule-based specifications). Каждое правило состоит из трёх частей. Первая часть — предусловие, содержит в себе предикаты, истинность которых меняется в зависимости от конфигурации системы. Если же все предикаты истинны, то правило можно применять. В этом случае происходит некоторое событие (тоже предикат), описанное во второй части правила, а также присваиваются истинные значения предикатам из постусловия. Очевидно, что такой способ спецификации аналогичен сетям Петри, правилам соответствуют переходы, пред- и постусловиям — входные и выходные места. Однако модифицированные раскрашенные сети Петри используются в работе [4] лишь для одного из оптимизированных алгоритмов выявления взаимодействий, основанного на инвариантах сетей Петри. Другой оптимизированный алгоритм использует симметрию частей системы. А третий алгоритм (EXH) — эквивалентен нашему, так как он использует граф достижимости системы. Размеры графов достижимости из нашей

работы сравнивались с соответствующими размерами, полученными в результате работы алгоритма EXH. Как и следовало ожидать, полученные графы достижимости имеют размер того же порядка, что и соответствующие графы в работе [4].

В качестве дальнейшего развития работы предполагается:

- развитие средств моделирования и верификации телефонных сетей;
- разработка автоматизированной системы для удобного добавления и удаления функциональностей;
- оптимизация методов верификации с учётом особенностей проблемной области.

СПИСОК ЛИТЕРАТУРЫ

1. Jensen K. Coloured Petri Nets: basic concepts, analysis methods and practical use. — Springer, 1996. — (Monographs on Theoretical Computer Science). — Vol. 1–3.
2. Keck D.O., Kuehn P.J. The Feature and Service Interaction Problem in Telecommunications Systems: A Survey // IEEE Transactions on Software Engineering. — 1998. — Vol. 24, N 10.
3. Kozura V.E., Nepomniaschy V.A., Novikov R.M. Verification of Distributed Systems Modelled by High-level Petri Nets // Proc. of the Internat. Conf. on Parallel Computing in Electrical Engineering. — Warsaw, Poland, IEEE Computing Society, 2002. — P. 61–66.
4. Nakamura M. Design and Evaluation of Efficient Algorithms for Feature Interaction Detection in Telecommunication Services: PhD diss. — Osaka University, 1999.
5. Nepomniaschy V.A., Argirov V.S., Beloglazov D.M.et al. Modeling and verification of SDL specified distributed systems using high-level Petri nets // Proc. Workshop on Concurrency, Specification and Programming (CS&P'2004), Humboldt University, Berlin, 2004. — P. 100–111.
6. Кларк Э.М., Грамберг О., Пелед Д. Верификация моделей программ: Model Checking. — М., Изд. МЦНМО, 2002.
7. Козюра В.Е., Непомнящий Н.А., Новиков Р.М. Верификация раскрашенных сетей Петри методом проверки моделей. — Новосибирск, 2001. — (Препр. / ИСИ СО РАН; № 89).
8. Непомнящий В.А., Алексеев Г.И., Аргиров В.С. и др. Программный комплекс SPV для симуляции, анализа и верификации спецификаций коммуникационных протоколов // Тр. Всеросс. научной конф. «Методы и средства обработки информации» (МСО-2003). — М.: МГУ, 2005. — С. 407–413.

Е. Ю. Ботоева

ДВУХ- И ТРЕХМЕРНАЯ ВИЗУАЛИЗАЦИИ МНОЖЕСТВА РЕШЕНИЙ В СИСТЕМЕ UNICALC

ВВЕДЕНИЕ

Система UniCalc [10] представляет собой среду для решения задач математического моделирования с удобным графическим интерфейсом. Математический аппарат системы UniCalc основан на недоопределенных вычислениях [1], что позволяет находить множество решений для произвольной системы ограничений (уравнений, неравенств, булевских утверждений). Также, имеются следующие модули, повышающие эффективность работы системы: модуль символьного преобразования, модуль для решения линейных ограничений и поиска точных корней.

Отличительной особенностью системы UniCalc является корректность получаемого решения несмотря на ограниченную точность вычислений на ЭВМ. Поэтому решение имеет вид набора интервалов, каждый из которых образован наименьшим и наибольшим возможным значением соответствующей переменной. Поясним на примере: пусть задана система ограничений y<=x; y>=x/2; y<=3–x. Множество решений этой системы показано на рис. 1.

Ответ, выданный системой (с включенным модулем для решения линейных ограничений [3]), будет иметь вид:

$$y = [0, 1.5],\ x = [0, 2.00000000000000045].$$

Здесь видно, что значение переменной x лежит в более широком интервале, чем на самом деле — в этом выражается учет ошибок округления, благодаря чему ни одно возможное решение не теряется. Для того чтобы найти точные решения (система выдала только прямоугольник, содержащий все возможные решения вместе другими точками), можно запустить поиск точных корней. Каждый такой корень будет находиться в этом прямоугольнике и будет являться решением.

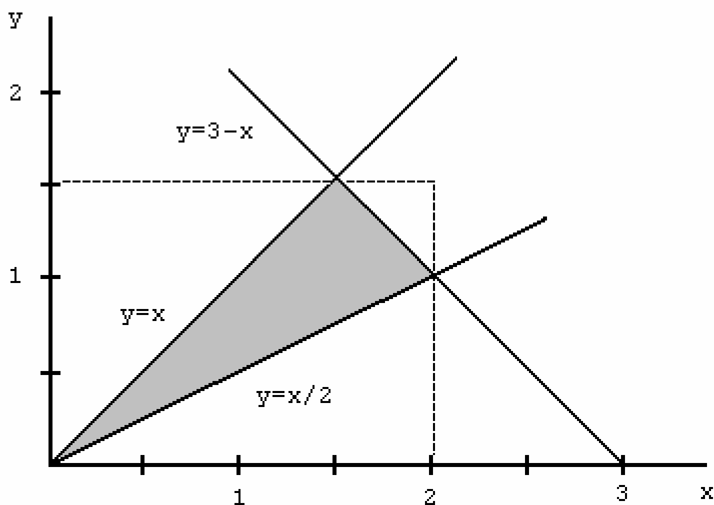


Рис. 1. Решение, представленное графически. Пунктиром выделена область, отвечающая решению, полученному системой UniCalc

Рассмотренный пример показывает, что решение, выраженное в интервалах значений, не дает всей полноты картины. Также ее не даст поиск точных корней — записанные кортежи значений так и будут только цифрами, в то время как на графике можно легко увидеть строение множества решений и в соответствии с этим, сделать выводы, внести нужные изменения в систему ограничений. Наглядность и удобство использования графиков при решении задач моделирования явились причиной создания графического модуля для системы UniCalc.

При работе над модулем, традиционно для системы UniCalc, много внимания уделялось корректной визуализации (более подробно о том, что значит корректность и для чего она нужна, смотрите в следующих разделах). В наши задачи входило создание методов и средств как двумерной, так и трехмерной корректной визуализации.

Существует ряд пакетов программного обеспечения для корректной визуализации систем ограничений с двумя переменными, например [4, 5, 7]. Насколько известно автору, пакетов для трехмерной корректной визуализации не существует. Таким образом, графический модуль для системы UniCalc, по-видимому, является первым таким пакетом.

Статья построена следующим образом. В разд. 1 мы вводим основные понятия, во втором — объясняем, для чего нужна корректность. Далее описываем алгоритм построения корректного приближения графика и получения изображения на экране. В заключение обсуждаем дальнейшие планы по развитию графического модуля.

ОСНОВНЫЕ ПОНЯТИЯ И ОПРЕДЕЛЕНИЯ

Определим понятия, которыми мы будем пользоваться в статье. Основными понятиями для нас будут график и покрытие.

Пусть $F(x_1,\ldots,x_n)$ — система ограничений, содержащая переменные x_1, …, x_n. В зависимости от контекста мы будем понимать $F(x_1,\ldots,x_n)$ либо как множество отдельных уравнений и неравенств, заданных пользователем системы UniCalc, либо как их логическую конъюнкцию.

Под *графиком системы ограничений* $F(x_1,\ldots,x_n)$ мы будем понимать множество ее решений:

$$\{ (a_1, \ldots, a_n) \mid \text{система ограничений } F(x_1,\ldots,x_n) \text{ выполняется при } x_1 = a_1, \ldots, x_n = a_n \}.$$

График — это математически точное представление множества решений системы ограничений, будем также говорить реальный график. Точки графика могут складываться в поверхности (кривые), могут образовывать разрозненные множества, при этом на графике отображается зависимость переменных друг от друга, и одну из них можно считать значением функции от остальных, в том числе заданной неявно.

Так как в системе UniCalc значения всех переменных ограничены по модулю «максимальным вещественным числом» maxReal, мы рассматриваем график из области, ограниченной множеством [–maxReal, maxReal] x … x [–maxReal, maxReal].

Строить мы будем некоторое приближение к реальному графику — покрытие. В нашем случае *покрытие* — это множество параллелепипедов (n-мерных), каждый из которых может содержать часть графика, а их объединение — весь график. Покрытие можно считать грубым представлением графика, при этом в нем может не оказаться ни одной точки графика (если система, заданная пользователем, несовместима, т. е. не имеет решений; это обусловлено особенностями машинной арифметики), но если существует хотя бы одно решение, то оно лежит в покрытии.

Наличие точек графика внутри параллелепипеда проверяется при помощи алгоритма недоопределенных вычислений (АНВ) [1]. Для произволь-

ной системы ограничений АНВ строит параллелепипед, содержащий все ее решения (и возможно, другие точки). Если построенный параллелепипед пуст, это означает, что система ограничений несовместна.

Минимальной составляющей покрытия является параллелепипед. Пусть параллелепипед определен следующим образом: I_1 x … x I_n, где I_k — проекция параллелепипеда на ось x_k. Тогда параллелепипед принадлежит покрытию, если он может содержать точки реального графика. Это условие проверяется при помощи АНВ, а именно проверяется, что для следующей системы ограничений:

$$F(x_1, \ldots, x_n) \cup \{ x_1 \in I_1, \ldots, x_n \in I_n \}$$

АНВ выдает непустой параллелепипед.

Для лучшего приближения покрытия к графику, размер параллелепипедов уменьшается — это называется процессом *детализации* покрытия. Если АНВ не может обнаружить несовместность системы ограничений, то это обязательно произойдет в процессе детализации покрытия. Подобный метод — уменьшение размеров покрывающих элементов для повышения точности результата — использовался в работе [11].

Кроме того, определим следующие понятия: *визуализация* — представление в графическом виде множества решений на компьютере, и результат этого представления; *изображение* — получаемая в результате визуализации всегда двухмерная картинка на экране монитора.

Решение задачи графическим способом — это получение графика системы ограничений. Как уже говорилось, мы будем визуализировать двухмерные и трехмерные графики, но переменных в задаче может быть произвольное число. Поэтому, фактически, мы будем заниматься визуализацией проекций графика на двухмерные и трехмерные пространства. Значит, значения выбранных переменных будут изменяться (переменные, соответствующие осям пространства проекции), значения остальных — некоторые фиксированные.

КОРРЕКТНАЯ ВИЗУАЛИЗАЦИЯ

Теперь подробно объясним, что мы понимаем под корректной визуализацией и зачем она нужна.

Рассмотрим способ построения изображения с помощью таблицы в двухмерном случае для ограничения $y = x \times \frac{x - 1.5 - 0.000001}{x - 1.5}$.

Это уравнение задает гиперболу, ветви которой практически касаются друг друга в точке x = y = 1.5, однако при x = 1.5 имеется разрыв.

Для каждого значения переменной x с некоторым шагом вычисляется значение переменной y, за шаг возьмем 1, начальное значение x — –3. С точностью до 10^{-5} эта таблица будет иметь следующий вид.

Таблица 1

Значения переменных x и y для заданного ограничения

x	–3	–2	–1	0	1	2	3
y	–3	–2	–1	0	1	2	3

Затем по этой таблице на координатной плоскости отмечаются точки и они соединяются отрезками. Результат этого построения представлен на рис. 2.

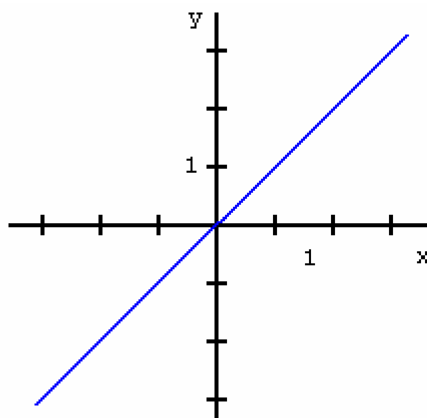


Рис. 2. Изображение, построенное по таблице

Очевидно, что построенное изображение не соответствует действительности (рис. 3) и может ввести в заблуждение, скрыв некоторые особенные точки (разрывы).

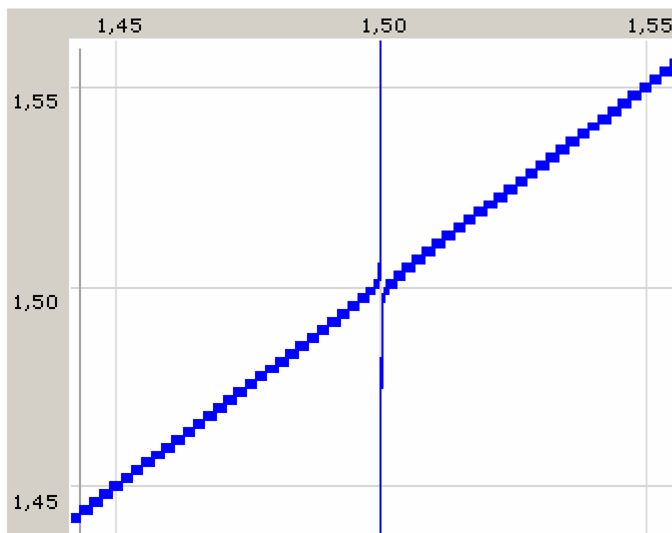


Рис. 3. Корректное изображение, полученное нашим модулем

Вообще, чтобы изображение лучше отражало реальную ситуацию, можно уменьшать шаг между контрольными точками. Но, во-первых, это не устранит проблемы, связанной с округлениями, а во-вторых, в общем случае особенные точки могут быть не обнаружены (заменим в ограничении 1.5 на √2).

Аппарат интервальной арифметики позволяет решить все эти проблемы. Покажем результат интервальных вычислений на другом примере: пусть задано ограничение $y = (10000 + \sin(x)) — 9999$. Построим таблицу со значениями y, посчитанными обычным и интервальным способами (почему получаются именно такие значения, см. [2]).

Таблица 2

Округленные значения y
(посчитанные с маленькой точностью — для наглядности)

x	-1	0	1	2	3
y_r	0	1	2	2	0

Таблица 3

Интервальные значения y

x	[-1,0]	[0,1]	[1,2]	[2,3]
y_i	[0.1585, 1]	[1, 1.841]	[1.841, 2]	[1.141, 1.909]

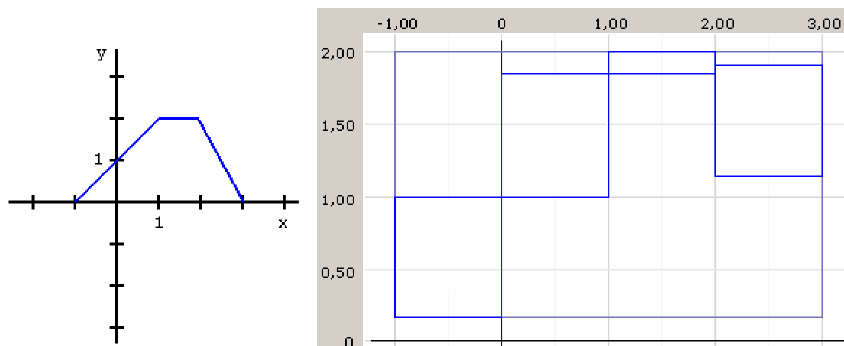


Рис. 4. Изображения, построенные разными способами.
Фактически, должно получиться изображение графика функции y = 1 + sin(x).
Первое — слишком грубое приближение

Из рис. 4. видно, что благодаря интервальным вычислениям строится изображение, соответствующее реальному графику. Уменьшая размер интервалов, получаемое изображение будет точнее, при этом оно остается корректным. Построение изображений с использованием интервальной арифметики легко реализуется в системе UniCalc, так как в ней все вычисления интервальные.

АЛГОРИТМ ПОСТРОЕНИЯ ПОКРЫТИЯ

Здесь описывается принцип построения покрытия — приближения к графику.

Изображение строится путем постепенной детализации уже полученного покрытия. Первое покрытие получается следующим образом: начальная оценка значений переменных с помощью АНВ дает нам для каждой из них внешние границы, которые образуют параллелепипед, он и является пер-

вым покрытием. Так мы сужаем все пространство до первого покрытия и рассматриваем точки только из него (вне этого параллелепипеда точек, удовлетворяющих системе ограничений, нет).

Затем идет пошаговая детализация покрытия. На каждом шаге из покрытия удаляется один параллелепипед и, если он может содержать точки реального графика, разбивается на 2^n равных, меньших параллелепипедов (n — размерность визуализации), которые добавляются в покрытие. За счет уменьшения размеров параллелепипедов приближение становится точнее. Этот процесс продолжается до тех пор, пока не будет достигнут нужный уровень детализации. На рис. 5 показаны несколько этапов работы этого алгоритма на примере двухмерной визуализации системы $x^2 + y^2 = 1$.

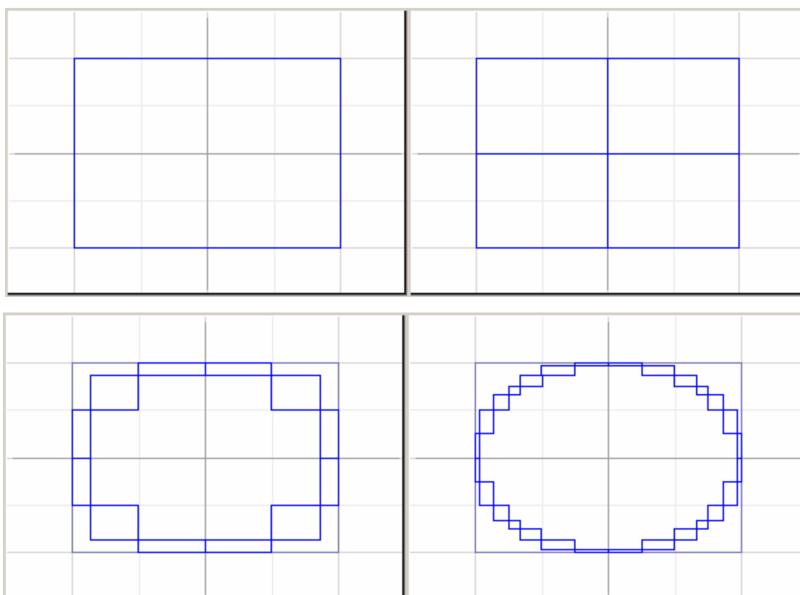


Рис. 5. Первые 4 покрытия для уравнения $x^2 + y^2 = 1$

Благодаря такому способу рисования мы достигаем следующие цели:

- на каждом шаге убираем из рассмотрения области пространства, заведомо не содержащие решений системы, и тем самым уменьшаем перебор ячеек пространства;
- всякий раз покрытие гарантированно содержит все решения системы, т. е. график является корректным отображением множества решений (вместе с избыточными точками).

ПОЛУЧЕНИЕ ИЗОБРАЖЕНИЯ В ДВУХМЕРНОМ СЛУЧАЕ

В двухмерном случае изображение состоит из прямоугольников. Каждому прямоугольнику из покрытия соответствует прямоугольник на изображении (в стандартном понимании). Размер прямоугольника на изображении определяется размерами клиентского окна и координатами прямоугольника в покрытии. Для удобства анализа изображения на рабочей области отображается сетка, состоящая из параллельных осям X и Y линий, а на горизонтальной и вертикальной линейках показываются значения координат.

ПОЛУЧЕНИЕ ИЗОБРАЖЕНИЯ В ТРЕХМЕРНОМ СЛУЧАЕ

В этом разделе описывается использованный нами подход к визуализации трехмерных множеств решений. Объект, который нам надо изобразить на экране трехмерный, в то время как на экране есть только два измерения, и изображение должно давать четкое представление об объекте.

Отображение трехмерного объекта на двухмерное изображение происходит с помощью *камеры*, параметром которой является пирамида видимости (рис. 6). Здесь под камерой можно понимать некоторый объектив, у которого есть определенный угол обзора, откуда он начинает и перестает «видеть». Благодаря этому алгоритму [8, 9], который также называется алгоритмом отсечения по пирамиде видимости, трехмерные объекты отображаются на двухмерную плоскость с учетом перспективы (а соответственно, и объема). К тому же этот алгоритм позволяет легко реализовать поворот в пространстве и изменение масштаба, при этом изменяются только параметры камеры.

Мы рисуем множество решений в виде проволочной модели — набор кубиков, состоящих из ребер. Система координат графика называется мировой системой координат. Для отображения на экран, координаты объекта

преобразуются из мировой системы в систему координат камеры, определяемой ее параметрами. Поскольку единицей отображения в проволочной модели является ребро — линия, проведенная между двумя точками, таким объектом будем считать ребро. Когда координаты вершин ребра преобразованы в нее, происходит проверка, попадают ли они в область видимости камеры, и в соответствии с этим происходит отсечение «невидных» отрезков. Затем, уже двухмерные координаты изменяются пропорционально размерам клиентского окна и выводятся на экран.

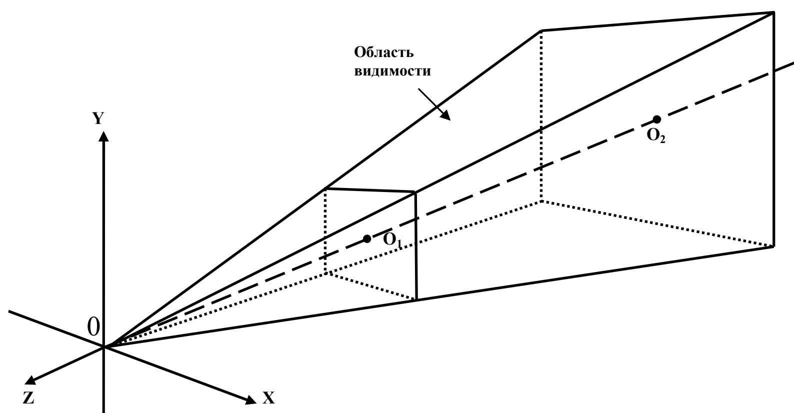


Рис. 6. Пирамида видимости.
На экран отображаются все объекты, попадающие в область видимости

Когда начинается отрисовка изображения, параметры камеры автоматически устанавливаются в «удобное» положение, которое охватывает весь график. Затем пользователь может приблизить/удалить изображение, повернуть вокруг вертикальной и горизонтальной оси.

Изображение представляет собой множество кубиков с нарисованными ребрами. Для придания ему большей наглядности, цвет ребер каждого кубика градуируется в соответствии с их направлением. Так, ребра, параллельные оси X, немного краснее, Y — зеленее, Z — синее. Удаления невидимых линий пока не происходит, и это создает трудности в анализе изображения. Реализация такого способа рисования, а также закрашивание граней запланированы на будущее. Получаемое таким способом изображение показано на рис. 7.

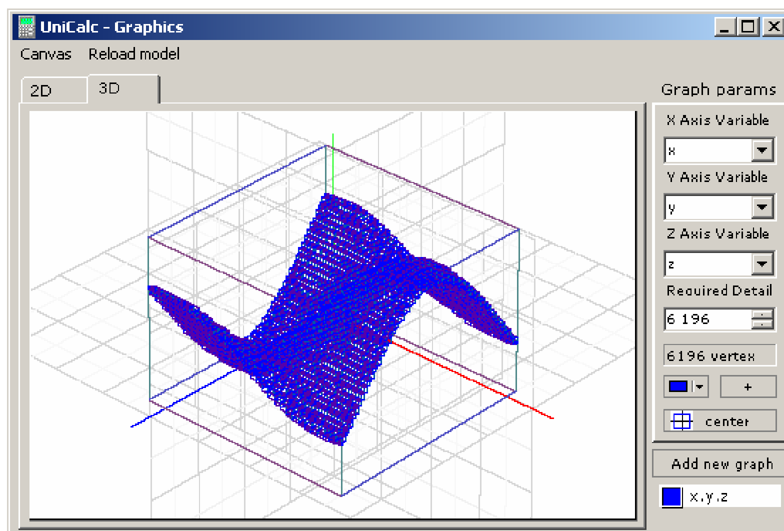


Рис. 7. Пример трехмерной визуализации для модели
x = [-2, 2]; z = [-2, 2]; y = sin(x) * cos(z)

ЗАКЛЮЧЕНИЕ

В системе UniCalc существует возможность увидеть графическое представление множества решений задачи (вместе с обычным, текстовым представлением). Визуализация решения представляет собой некоторое приближение реального графика в виде множества параллелепипедов — покрытия. Такое покрытие всегда гарантированно содержит все точки реального графика и, возможно, другие точки пространства. Точность покрытия можно повышать, тогда часть избыточных точек будет выброшена.

В настоящее время планируется дальнейшая работа по повышению удобства использования графического модуля и эффективности его работы. Способ представления изображения в том виде, как это делается сейчас, имеет некоторые недостатки, а именно, проволочная модель без удаления невидимых линий затрудняет анализ поверхности. Возможны два пути решения: удаление невидимых линий в проволочной модели, либо раскрашивание граней параллелепипедов с использованием z-буфера или других методов. Предпочтительной для нас является полная раскраска граней, а по-

сле выполнения этой задачи можно будет приступить к градации цвета в зависимости от кривизны поверхности.

Также интерес представляет функциональность, связанная с отсечением полупространств, заданных некоторыми плоскостями, определением координат точек, принадлежащих покрытию, локализацией решений и прочее. Реализация всех этих возможностей даст нам наглядный и удобный инструмент для анализа множества решений задачи.

СПИСОК ЛИТЕРАТУРЫ

1. **Нариньяни А.С., Телерман В.В., Ушаков Д.И., Швецов И.Е**. Программирование в ограничениях и недоопределенные модели // Информационные технологии. — 1998. — №7.
2. **Алефельд Г., Херцбергер Ю.** Введение в интервальные вычисления. — М.: Мир, 1987.
3. **Петров Е.С., Костов Ю.В., Ботоева Е.Ю.** Модуль для решения линейных ограничений в системе UniCalc // Тр. VIII междунар. конф. «Проблемы управления и моделирования в сложных системах» / Под ред. акад. Федосова Е.А., акад. Кузнецова Н.А., акад. Виттиха В.А. — Самара: Самарский научный центр РАН, 2006. — 572с. ISBN 5-93424-227-X.
4. **Bossé M., Nandakumar N. R.**, When Equalities Are Not Equal: Missing Mathematical Precision in Teaching, Texts, and Technology // The College Mathematics Journal. — 2003. — Vol. 34, N 5.
5. **Reliable** Two-Dimensional Graphing Methods for Mathematical Formulae with Two Free Variables. SIGGRAPH 2001: Proc. / 28th Annual Conf. on Computer Graphics, Los Angeles, California, USA. ACM, 2001. — Los Angeles, 2001. — 1-12 p.
6. **Shou H., Shen J., Yoon D.**, Robust plotting of polar algebraic curves, space algebraic curves and offsets of planar algebraic curves. — Reliable Computing, 2005.
7. **Hickey T. J., Zhe Qiu, van Emden M. H.** Interval Constraint Plotting for Interactive Visual Exploration of Implicitly Defined Relations. — Reliable Computing, 2000.
8. **Фокс А., Пратт М**. Вычислительная геометрия. Применение в проектировании и на производстве. Пер. с англ. — М.: Мир, 1982.
9. **Тихомиров Ю**. Программирование трехмерной графики — СПб.: БХВ—Санкт-Петербург, 1999. — 256 с.
10. **Костов Ю.В., Липовой Д.А., Мамонтов П.Г., Петров Е.С**. Новый UniCalc: версия 5 — возможности и перспективы // Тр. IX национальной конф. по искусственному интеллекту. — КИИ-2004. — Тверь, 2004. — 915–922 с.
11. **Кашеварова Т.П**. Использование системы UniCalc для решения задач математического моделирования. — Новосибирск, 1999. — 18-22 с. — (Препр. / СО РАН. Ин-т систем информатики; № 64).

С. А. Бражник

ФОРМАЛЬНАЯ МОДЕЛЬ ДИАГРАММЫ КЛАССОВ ЯЗЫКА UML[1]

1. ВВЕДЕНИЕ

Унифицированный язык моделирования UML [1, 2] широко применяется в сфере промышленного производства программного обеспечения. Тем не менее в спецификации UML существует еще множество огрехов и противоречий [3, 4], которые без применения некоторого формального подхода будет сложно разрешить.

Многие научные исследования, посвященные формализации модели и метамодели языка UML, основываются не на самом UML, а на некотором его подмножестве — формальном и строго структурированном. Например, авторы [5] предложили формальную спецификацию метамодели объектно-ориентированного языка моделирования BON. Но BON в сравнении с UML более формализован и «подогнан» под условия решаемой задачи.

Схожий подход использован в работе [6]. Продемонстрирована применимость подхода к формальной спецификации языка UML. При этом формализуется только часть спецификации. Стоит также отметить, что предлагаемое решение ограничено представлением всех метауровней моделируемой системы в одной модели, что при большом объеме моделируемой системы может стать проблемой.

В работе [7] автор демонстрирует применимость алгебраического метода для формального описания ER-диаграмм, являющихся аналогом диаграмм классов UML. Этот подход является наиболее близким подходу, предложенному в данной работе.

Цель этой работы — формализовать диаграмму классов языка UML. При построении формальной модели диаграммы использованы простейшие понятия теории множеств и многоосновных алгебр. Основой подхода является использование машин абстрактных состояний (МАС) Ю. Гуревича [8] и динамические системы с неявным состоянием [10].

Далее статья организована следующим образом: в разд. 2 описывается репрезентативный пример, следующий раздел посвящен построению схемы классов, ее иерархической корректности и замыканию схемы, разд. 4 со-

[1] Работа поддержана грантом РФФИ № 04-01-00272.

держит описание построения алгебры программы, а в заключительном разделе обсуждаются достоинства и перспективы этой работы.

2. РЕПРЕЗЕНТАТИВНЫЙ ПРИМЕР

Репрезентативный пример, который мы будем использовать далее в этой работе, состоит из двух диаграмм. Первая диаграмма представляет собой структуру проекта и служит нам для демонстрации интерфейсов и их взаимодействий в программе (рис. 1).

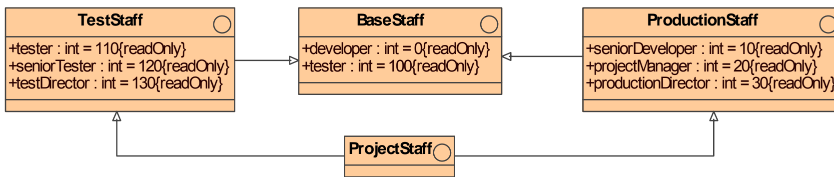


Рис. 1. Структура проекта

Вторая диаграмма иллюстрирует иерархию классов ролей пользователей в системе (рис. 2)

3. ОСНОВНЫЕ ПОНЯТИЯ

3.1. Система типов

Модель базируется на системе типов со следующей грамматикой:

$$T ::= BASE \mid CLASS \mid INTERF,$$

где *BASE*, *CLASS* и *INTERF* — три непустые непересекающиеся множества. *BASE* — множество базовых типов, *CLASS* — множество имен классов, *INTERF* — множество имен интерфейсов.

Элементы *T* называются *типовыми выражениями* или *типами*.

T^* — множество всех последовательностей элементов *T*, включая пустую последовательность <>.

2^{CLASS} — степень множества *CLASS*, а $T^{v} = T \cup \{void\}$, где *void* — специальный тип, не принадлежащий *T*.

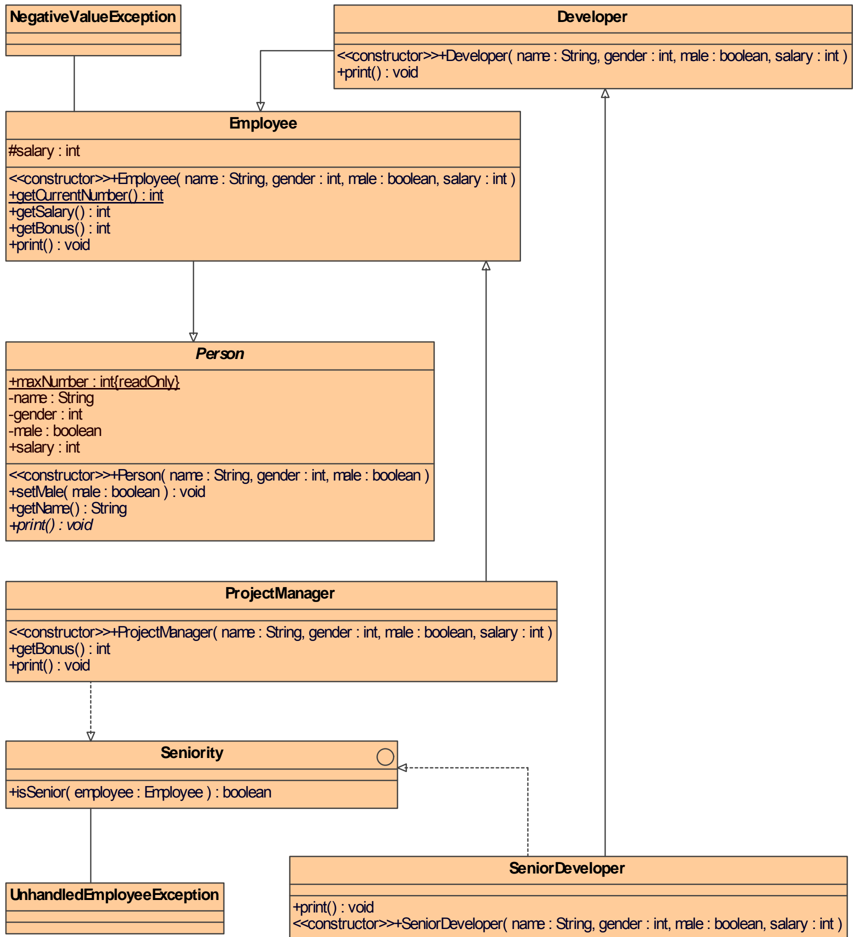


Рис. 2. Иерархия ролей в системе

3.2. Схема классов

Множество объявлений классов и интерфейсов мы называем *схемой классов*. Пусть *ATT* и *METH* являются конечными непустыми множествами

имен полей и *имен методов* соответственно, а $c \in CLASS$ и $i \in INTERF$ — именами класса и интерфейса.

Тогда *собственное объявление класса c* — это четыре следующих частичных функций:

$var(c) : ATT \to T$ $const(c) : ATT \to T$

$con(c) : T^* \to 2^{CLASS}$ $meth(c) : METH \times T^* \to 2^{CLASS} \times T^v$,

представляющих соответственно: объявление переменных, объявление констант, объявление конструкторов и объявления методов.

Заметим, что переменная класса в языке UML [1] может быть представлена в виде атрибута класса или же как отношение ассоциации с классом некоторого типа. В данной работе мы рассмотрим только первый вариант.

Собственное объявление интерфейса i — это пара частичных функций:

$const(i) : ATT \to T$ $meth(i) : METH \times T^* \to 2^{CLASS} \times T^v$.

Например, собственное объявление класса *Person* выглядит следующим образом:

```
var(Person) = {(name → String), (gender → int), (male → boolean),
               (salary → int)}
const(Person) = {(maxNumber → int)}
con(Person) = {<name, gender, male> → 0}
meth(Person) = {(setMale,<boolean> → <0, void>), (getName,<> →
               <0, String>)}.
```

Собственное объявление интерфейса *ProductionStaff* — выглядит так:

```
const(ProductionStaff) = {(seniorDeveloper → int),
              (projectManager → int), (productionDirector → int)};
meth(ProductionStaff) = {}.
```

Каждое объявление переменной или константы вводит ее имя *x* и тип *t*, каждое объявление конструктора — строку типов параметров *r* и множество типов исключений *q*, каждое объявление метода — имя метода *m*, строку типов параметров *r*, множество типов исключений *q* и тип результата *t*.

Далее мы будем обозначать элементы *var* и *const* парами (*x, t*), элементы *con* — парой (*r, q*), элементы *meth* — четверкой (*m, r, t , q*).

Пусть *CDECL* обозначает множество собственных объявлений класса, а *IDECL* — множество собственных объявлений интерфейса. Тогда *схема классов S* состоит из следующих элементов:

- конечное подмножество *I* множества *INTERF;*

- конечное подмножество *C* множества *CLASS;*
- множество типов исключений $Q \subset C$;
- бинарное ациклическое отношение *isaI* на *I*;
- бинарное ациклическое отношение *isaC* на *C*;
- бинарное отношение *impl* на $C \times I$;
- две тотальные функции $cdecl : C \to CDECL$ и $idecl : I \to IDECL$,

так что:

для каждого используемого типа t выполняется условие:
$t \in (BASE \cup I \cup C)$,
для любого $ic \in (I \cup C)$ и $(m, r, t, q) \in meth(ic)$ и $(r,q) \in con(ic)$ соблюдается $q \subseteq Q$,
если $c \in C$ и $x \in ATT$, то может существовать либо $(x, t) \in var(c)$, либо $(x, t') \in const(c)$.

Отношение *isaC* и *isaI* определяет *граф множественного наследования* на множестве классов *C* и интерфейсов *I* соответственно. Отношение *impl* определяет *граф реализаций*, в котором имя класса может быть связано с одним или несколькими именами интерфейсов. Тогда для каждого имени класса *c* в *C*, кортеж *(c, isaC(c), impl(c), var(c), const(c), con(c), meth(c))* — это *полное объявление класса* с именем *c* и также для интерфейса *i* в *I* кортеж *(i, isaI(i), const(i), meth(i))* — *полное объявление интерфейса* с именем *i*.

Для репрезентативного примера указанные отношения и множества выглядят следующим образом:

```
I = {BaseStaff, TestStaff, ProductionStaff, ProjectStaff, Seniority}
C = {Person, Employee, Developer, SeniorDeveloper, ProjectManager,
      NegativeValueException, UnhandledEmployeeException}
isaI = {(TestStaff, BaseStaff), (ProductionStaff, BaseStaff),
      (ProjectStaff, TestStaff), (ProjectStaff, ProductionStaff)}
isaC = {(Employee, Person), (Developer, Employee), (ProjectManager,
      Employee), (SeniorDeveloper, Developer)}
impl = {(SeniorDeveloper, Seniority), (ProjectManager, Seniority)}.
```

Из приведенных определений непосредственно вытекает следующее.

1. Модель поддерживает множественное наследование классов и интерфейсов, а также множественные реализации интерфейсов.

2. Не допускается совмещение имен полей в пределах класса или интерфейса, в то время как допускается совмещение имен методов при условии различности их сигнатур.

Заметим, что определенная нами схема классов является аналогом сигнатуры в алгебрах и, как следствие, не содержит тел методов и конструкторов. Рассмотрение данной темы выходит за рамки этой статьи.

3.2.1. Отношение тип—подтип

Каждая схема естественным образом определяет строгий порядок над классами и интерфейсами, который мы называем отношением *тип—подтип* и обозначаем $<_{isa}$. Оно определяется рекурсивно следующим образом: если $ic <_{isa} ic'$ и $ic' <_{isa} ic''$, тогда $ic <_{isa} ic''$.

Также будет применяться запись $ic \leq_{isa} ic''$ (частичный порядок), чтобы учесть случай $ic = ic''$.

Например:
`ProjectStaff` $<_{isa}$ `ProductionStaff`, `ProjectStaff` $<_{isa}$ `BaseStaff`, `ProjectManager` $<_{isa}$ `Seniority`, `ProjectManager` $<_{isa}$ `Employee`, `ProjectManager` $<_{isa}$ `Person`.

3.2.2. Классификация полей и методов

Компоненты классов подразделяются на несколько пересекающихся групп согласно ряду ортогональных классификаций.

Так, для любого класса *c* множество *var(c)* состоит из двух непересекающихся подмножеств *cvar(c)* и *ovar(c),* называемых *переменными класса* и *переменными объекта* соответственно (любое из подмножеств может быть пустым). Подобным же образом множество *meth(c)* состоит из двух непересекающихся подмножеств *cmeth(c)* и *ometh(c),* называемых *методами класса* и *методами объекта*.

Так, для `cmeth(Employee) = {(getCurrentNumber, <>, int, 0)}` и `ometh(Employee) = {(getSalary, <>, int, 0), (getBonus, <>, int, NegativeValueException)}`.

По другой классификации в классе *c* может существовать подмножество методов *ameth(c)*, называемое *абстрактными методами*. Согласно требованиям языка *cmeth(c) \ ameth(c) = 0.*

Абстрактный метод может быть объявлен только в абстрактном классе, где он помечается как abstract. Метод, объявленный в интерфейсе *i*, является абстрактным по определению, так что $ameth(i) = meth(i)$.

По третьей классификации компоненты класса подразделяются на открытые, защищенные пакетные и приватные компоненты с различными правами доступа. Таким образом, множества переменных, констант, конструкторов и методов в классе *c* могут быть представлены следующими непересекающимися множествами:

$const(c) = privconst(c) \cup packconst(c) \cup protconst(c) \cup publconst(c),$

$var(c) = privvar(c) \cup packvar(c) \cup protvar(c) \cup publvar(c),$

$con(c) = privcon(c) \cup packcon(c) \cup protcon(c) \cup publcon(c),$

$meth(c) = privmeth(c) \cup packmeth(c) \cup protmeth(c) \cup publ(c)$

при следующем условии: $ameth(c) \cap privmeth(c) = 0$.

3.2.3. Скрытие, подмена и наследование

Компонент суперкласса может быть скрыт, подменен или наследован. Будем говорить, что поле (x, t) класса/интерфейса ic' близко классу/интерфейсу *ic*, если $ic <_{isa} ic'$, (x, t) — неприватное поле в ic' и нет такого класса/интерфейса ic'' и типа t', что $ic <_{isa} ic'' <_{isa} ic'$ и $(x, t') \in (var(ic'') \cup const(ic''))$. Точно также метод (m, r, t, q) класса/интерфейса ic' близок классу/интерфейсу *ic*, если $ic <_{isa} ic'$, (m, r, t, q) — неприватный метод в ic' и нет такого класса/интерфейса ic'' типа t' и множества имен классов q', что $ic <_{isa} ic'' <_{isa} ic'$ и $(m, r, t', q') \in meth(ic'')$. Таким образом, поле, близкое некоторому классу/интерфейсу, объявлено в каком-то его суперклассе/суперинтерфейсе и не переобъявлено ни в каком промежуточном классе/интерфейсе, то же самое относится и к методам. Например, константа *maxNumber* класса *Person* близка классу *Developer*, потому что нет объявления поля с таким же именем в промежуточном классе *Employee*. А вот метод *getBonus* класса *Employee* не будет близок какому-либо подклассу класса *ProjectManager*, так как этот метод переобъявлен в последнем.

- Поле (x, t') класса/интерфейса ic' близкое классу/интерфейсу *ic* скрыто в *ic*, если существует $(x, t) \in (var(ic) \cup const(ic))$. Класс/интерфейс *ic* наследует близкое ему поле (x, t') класса/интерфейса ic', если не существует $(x, t) \in (var(ic) \cup const(ic))$.
- Метод $(m, r, t', q') \in cmeth(c')$ (т.е. метод класса) близкий классу *c* считается скрытым в *c*, если существует $(m, r, t, q) \in cmeth(c)$, а

метод объекта $(m, r, t', q') \in ometh(ic')$ близкий классу/интерфейсу ic считается подмененным (реализованным) в классе/интерфейсе ic, если существует $(m, r, t, q) \in ometh(ic)$.

- Класс/интерфейс ic наследует близкий ему метод (m, r, t', q') класса/интерфейса ic', если не существует $(m, r, t, q) \in ometh(ic)$.

У нас, поле *salary* класса *Person* скрыто в классе *Employee*, метод *getName* первого наследуется в последнем, а метод *print* подменен в каждом из подклассов класса *Person*.

3.3. Иерархическая корректность

Мы говорим, что схема *S* иерархически корректна, если выполняется следующее:

- все подменяемые/наследуемые/реализуемые методы должны иметь один и тот же тип результата и не могут принадлежать к непересекающимся группам;
- при множественном наследовании абстрактных методов (что может иметь место как в интерфейсах, так и в классах) наследуемые объявления методов должны быть идентичными;
- при множественном наследовании методов наследуемые объявления методов должны быть идентичными;
- подменяющий/скрывающий метод не может возбуждать исключения, не объявленные в подменяемом/скрываемом методе;
- если конкретный класс наследует метод, объявленный в интерфейсе, то этот метод не может возбуждать исключения, не объявленные в реализуемом методе;
- метод, подменяющий или скрывающий открытый метод, должен быть открытым;
- метод, подменяющий или скрывающий защищенный метод, должен быть либо открытым, либо защищенным;
- подменяющий или скрывающий пакетный метод не должен быть приватным;
- если класс наследует абстрактный метод и при этом не объявляет и не наследует конкретный метод с той же самой сигнатурой, то он должен быть абстрактным;
- каждый абстрактный метод должен быть реализован в каждом из конкретных подклассов.

3.4. Замыкание схемы

Для любого класса/интерфейса *ic* и поля *x* мы определим частичную функцию *ResF(ic, x)*, вырабатывающую имя класса/интерфейса, в котором поле *x* объявлено или из которого оно может быть однозначно наследовано. Если *ResF(ic, x) = ic'*, то либо *ic' = ic* и *x* объявлено в *ic*, либо $ic \neq ic'$ и *x* наследуется из *ic'*. Поэтому, если *(x, t)* — объявление переменной в *ic'*, мы можем расширить множество *var(ic)* полем *(x, t)*, определив $\overline{var}(ic) = (x, t)$, и если *(x, t)* — объявление константы в *ic'*, мы можем расширить множество *const(ic)* константой *(x, t)*, определив $\overline{const}\,(ic) = (x, t)$. Таким образом,

$$\frac{ResF(ic,x) = ic' \quad (x,t) \in var(ic')}{(x,t) \in \overline{var}(ic)} \qquad \frac{ResF(ic,x) = ic' \quad (x,t) \in const(ic')}{(x,t) \in \overline{const}(ic)}.$$

Для любого класса/интерфейса *ic*, имени метода *m* и строки типов *r* применяется тот же подход.

Например, множество *var(Employee)* расширяется элементом *(name, string)*, множество *const(ProductionStaff)* — элементами *(developer, int)*, *(tester, int)* и т.д. Заметим, что при этом множество *const(ProjectStaff)* не будет расширено константой *tester*, так как она не наследуются однозначно.

Множество *meth(SeniorDeveloper)* расширяется элементами *(getBonus, <>, int, NegativeValueException)*, *(setMale, <boolean>, void, 0)* и *(getCurrentNumber, <>, int, 0)*.

Подмножества множеств *var(c)*, *const(ic)*, *meth(ic)* каждого класса *c* и каждого класса/интерфейса *ic* расширяются соответствующим образом, чтобы включить наследуемые компоненты. Полученную схему классов мы называем замыканием.

Можно доказать, что если схема классов *S* иерархически корректна, то иерархически корректно и ее замыкание $\overline{S}$. Далее мы будем рассматривать, только иерархически корректные схемы.

4. ПОСТРОЕНИЕ АЛГЕБРЫ ПРОГРАММЫ

4.1. Базисная алгебра

Будем считать, что *базисная алгебра B* связывает с каждым типом *t* некоторое множество, его *носитель*, и с каждой операцией типа *t* — частичную функцию, ее *реализацию*. Носители наших типов определяются следующим образом:

- носитель базисного типа t — это некоторое множество B_t;
- носитель каждого класса $c \in C$ — специальное множество *Reference*, элементы которого называются ссылками;
- носитель типа *void* — одноэлементное множество $\{\perp\}$.

Носители всех типов — непересекающиеся множества. Существует также специальное значение null, не принадлежащее никакому носителю.

4.2. Состояния программы

Любая программа в различные моменты времени может обладать различными состояниями.

Алгебра состояния A программы *P* схемы $\overline{S}$ определяется следующим образом.

1. $A_t = B_t$ для любого базисного типа *t* и $op^A = op^B$ для каждой операции/константы базисного типа.
2. Конечное множество $A_{ic} = A_{ic}^o \cup \{null\}$, где $A_{ic}^o \subset Referece$, связывается с каждым $ic \in (I \cup C)$, так что если $ic <_{isa} ic'$, то $A_{ic}^o \subseteq A_{ic'}^o$, иначе оба условия $o \in A_{ic}^o$ и $o \in A_{ic'}^o$ выполняются тогда и только тогда, когда существует такой ic'', что $ic'' <_{isa} ic$, $ic'' <_{isa} ic'$ и $o \in A_{ic''}^o$.

   В каждом состоянии у каждого класса, интерфейса есть множество ссылок, так что множество ссылок супертипа включает в себя множества ссылок всех его подтипов и множества не пересекаются, если их типы не связаны отношением тип-подтип и не имеют общих подтипов. Например, $A_{SeniorDeveloper} \subset A_{Seniority}$, $A_{SeniorDeveloper} \subset A_{Developer} \subset A_{Employee}$.
3. Частичная функция $x_{ct}^A : A_c \to A_t$ связывается с каждой переменной объекта $(x,t) \in \overline{ovar(c)}$ таким образом, что для каждой пары классов (c, c') если $c' <_{isa} c$, c' наследует x и $o \in A_{c'}$, то $x_{ct}^A(o) = x_{c't}^A(o)$. Эта функция не определена только на null.

   Например, функция $name_{Employee,String}$ наследует часть функции $name_{Person,String}$, так что независимо от того, рассматривается ли сотрудник *o* как объект класса *Employee* или как объект класса *Person*, оба вызова функции $name_{Employee,String}(o)$ и $name_{Person,String}(o)$ вырабатывают один и тот же результат.

4. Алгебраическая константа $x_{ct}^{A} : A_t$ связывается с каждой переменной класса $(x, t) \in \overline{cvar}(c)$.

Статическая переменная представляется алгебраической константой. Она может иметь разные значения в разных состояниях.

5. Алгебраическая константа $y_{ict}^{A} : A_t$ связывается с каждой константой класса $(y, t) \in \overline{const}(ic)$ таким образом, что для каждой пары классов/интерфейсов *(ic, ic')* если $ic' <_{isa} ic$ и *ic'* наследует *y*, то $y_{ict}^{A} = y_{ic't'}^{A}$.

Константа, объявленная в схеме классов, также представляется алгебраической константой, но при этом остается одной и той же в каждом состоянии и наследуется всеми подтипами класса/интерфейса, в котором она объявлена. Например, константа *maxNumber*, объявленная в *Person*, будет иметь то же самое значение во всех его подклассах и будет сохранять это значение во всех состояниях программы.

У различных состояний программы данной схемы классов *S* сохраняется одна и та же базисная алгебра. Далее $state_B(S)$ обозначает множество всех состояний программы схемы *S* с базисной алгеброй *B*, а любое конкретное состояние из этого множества называется S_B-состоянием.

4.3. Обновление состояния

Одно состояние может быть преобразовано в другое посредством одного из *обновлений состояния* — *обновления функции* или *обновление носителя*.

Обновление функции в S_B-состоянии *A* — это тройка $(f_{rt}, \overline{o}, v)$, где f_{rt} — имя функции *f* со строкой типов параметров $r = t_1, ..., t_n$ и типом результата *t*, $\overline{o} = o_1, ..., o_n$ — элементы множеств $A_{t_1}, ..., A_{t_n}$, соответственно, и *v* — элемент множества A_t.

Обновление функции $\partial = (f, \overline{o}, v)$ служит для преобразования S_B-состояния *A* в новое S_B-состояние $A\partial$ следующим образом:

- $g^{A\partial} = g^{A}$ для любой функции *g*, отличной от *f*;
- $f^{A\partial}(\overline{o}) = v$;
- $f^{A\partial}(\overline{o}') = f^{A}(\overline{o}')$ для кортежа $\overline{o}'$, отличного от $\overline{o}$;

- $(A\partial)_c = A_c$ для любого $c \in (I \cup C)$.

Мы говорим, что состояние *A* преобразуется в состояние $A\partial$ путем запуска обновления ∂ [8]. *f* — это имя переменной, запуск обновления ∂ в *A* меняет либо значение переменной класса ($\overline{o}$ =<>), либо значение переменной объекта ($\overline{o}$ =<*o*>).

Обновление носителя μ в *A* — это пара *(c, o)*, где *c* — имя класса, а *o* — ссылка, такая что $o \in Reference$ и $o \notin |A|$, где |*A*| — множество всех носителей состояния *A*.

Обновление носителя μ =*(c, o)* преобразует S_B-состояние *A* в новое состояние $A\mu$ следующим образом:

- $(A\mu)_c = A_c \cup \{o\}$ и $(A\mu)_{c'} = A_{c'}$ для любого $c' \in (I \cup C)$, отличного от *c*;
- $f^{A\mu} = f^A$ для любой функции *f*.

Множество обновлений состояния *Г* называется *множеством обновлений*. Множество обновлений противоречиво, если оно содержит

- два таких обновления функции $\partial_1 = (f, \overline{o}, v)$ и $\partial_2 = (f, \overline{o}, v')$, что $v \neq v'$;
- два таких обновления состояния μ_1 =*(c, o)* и μ_2 =*(c', o)*, что $c \neq c'$.

Множество обновлений непротиворечиво в противном случае.

Непротиворечивое множество обновлений *Г*, примененное к S_B-состоянию *A*, преобразует его в состояние *A'* путем одновременного запуска всех обновлений $\partial \in \Gamma$ и $\mu \in \Gamma$. Если *Г* противоречиво, *A'* не определено. Если *Г* пусто, *A'* совпадает с *A*. Применение множества обновлений *Г* к состоянию *A* обозначается *AГ*.

Операция последовательного объединения двух непротиворечивых множеств обновлений Γ_1 и Γ_2, обозначаемая $\Gamma_1\Gamma_2$, производит непротиворечивое множество обновлений следующим образом: из $\Gamma_1 \cup \Gamma_2$ удаляется любое $\partial_1 \in \Gamma_1$, для которого существует такое $\partial_2 \in \Gamma_2$, что множество $\{\partial_1, \partial_2\}$ противоречиво.

Замыкание множества обновлений *Г* обозначается $\overline{\Gamma}$ и определяется следующим образом:

- если $(c, o) \in \Gamma$ и $c <_{isa} c'$, то $(c', o) \in \overline{\Gamma}$;

- если $(x_{ct}, o, v) \in \Gamma$, то $(x_{c't}, o, v) \in \overline{\Gamma}$ для всех *c'*, либо наследующих *x* из *c*, либо имеющих поле *x*, близкое *c*.

Fact 1. Замыкание непротиворечивого *Г* непротиворечиво.

Fact 2.

1. Для любого корректного состояния *A* и замкнутого множества обновлений *Г*, *AГ* — корректное состояние.

2. Если Γ_1 и Γ_2 — замкнутые множества обновлений, то таково и $\Gamma_1\Gamma_2$.

3. Для любого состояния *A* и множеств обновлений Γ_1 и Γ_2, $A(\Gamma_1\Gamma_2) = (A\,\Gamma_1)\,\Gamma_2$.

4.4. Представление программы

Обозначим через $\Gamma\Gamma^A(S)$ множество всех замкнутых множеств обновлений в данном состоянии программы *A* схемы *S*. Введем понятие пары *(Г, v)*, где *Г* — множество обновлений, а *v* — элемент некоторого носителя, и обозначим через $\Gamma\Gamma_t^A(S)$ множество пар *(Г, v)*, в которых $v \in A_t$ и $\Gamma \in \Gamma\Gamma^A(S)$.

Наконец, для любого S_B-состояния *A* и последовательности типов $r = t_1 ... t_n$ обозначим $A_{t_1} \times ... \times A_{t_n}$ через A_r, которое является одноэлементным множеством, когда *n* = 0.

Следующие компоненты составляют программу *P(B)* схемы *S*.

1. Подмножество |*P(B)*| множества состояний $state_B(S)$, называемое носителем программы.
2. Частичная функция $c_{cr}^A : A_c \times A_r \to \Gamma\Gamma_t^A(S)$ для каждого $c \in C$, $(r, q) \in con(c)$ и $A \in |P(B)|$, где $t \in (q \cup \{void\})$. Функция не определена для каждой пары $(null, v) \in A_c \times A_r$.
3. Частичная функция $m_{cr}^A : A_c \times A_r \to \Gamma\Gamma_{t'}^A(S)$ для каждого $c \in C$, конкретного метода $(m,r,t,q) \in \overline{ometh(c)}$ и состояния $A \in |P(B)|$, где $t' \in (q \cup \{t\})$, так что если $c' <_{isa} c$ и *c'* наследует *(m, r, t, q)*, то $m_{cr}^A(o,v) = m_{c'r}^A(o,v)$ для каждой пары $(o, v) \in A_{c'} \times A_r$. Функция не определена для каждой пары $(null, v) \in A_{c'} \times A_r$.

4. Частичная функция $m^A_{cr} : A_r \rightarrow \Gamma\Gamma^A_{t'}(S)$ для каждого $c \in C$, конкретного метода $(m,r,t,q) \in \overline{cmeth}(c)$ и состояния $A \in |P(B)|$, где $t' \in (q \cup \{t\})$, так что если $c' <_{isa} c$ и c' наследует (m,r,t,q), то $m^A_{cr}(v) = m^A_{c'r}(v)$ для каждого $v \in A_r$.

Программа обладает рядом состояний (п. 1). Для каждого конструктора заводится собственная функция, вырабатывающая множество обновлений и, возможно, значение, отличное от $\perp$, если в конструкторе возбуждается исключение (п. 2). Точно так же отдельная функция заводится для каждого метода, и такая функция вырабатывает пару, состоящую из множества обновлений и значения либо типа результата *t*, либо одного из типов исключений q (пп. 3 и 4). Значение типа *t* вырабатывается в случае нормального завершения метода, а значение типа исключения — в случае возбуждения методом исключения данного типа.

В п. 3 дополнительно указывается, что если метод объекта наследуется некоторым подклассом, то объект, принадлежащий и подклассу, и его суперклассу, снабжается одним и тем же методом. Заметим, что у функции, сопоставляемой методу класса, нет параметра-объекта, и потому такая функция наследуется полностью (п. 4).

Каждый из классов *Employee*, *Developer*, *ProjectManager* и *SeniorDeveloper* будет снабжен своей функцией *print* (подмена), одной и той же функцией для метода *getCurrentNumber* и одной и той же функцией для метода *getSalary* (наследование). Методы в абстрактных классах и интерфейсах остаются без реализаций.

Результатом работы является построение формальной модели диаграммы классов языка UML, а также определение ее семантики в стиле машин абстрактных состояний. Переход определенной нами машины из одного состояния в другое отражает операционную составляющую данного подхода.

В машине каждый объект обладает состоянием и ссылкой. Состояние объекта представляется рядом переменных объекта, значения которых могут быть обновлены. В модели эти переменные определены естественным образом как функции на множестве ссылок класса. Такая функция не определена на ссылке null. Состояние объекта может быть проанализировано или изменено методом, являющимся функцией, зависящей от состояния. Результат работы обновляющего метода — множество обновлений, являющееся строго определенной математической структурой. Поэтому се-

мантикой метода является множество функций, производящих множества обновлений.

3. ЗАКЛЮЧЕНИЕ

В работе предложен подход по формализации диаграммы классов языка UML. Главная задача заключалась в построении объектно-ориентированной операционной машины высокого уровня, элементы которой естественным образом отображались бы в диаграмму классов языка и наоборот. Подход базируется на семантическом представлении в стиле машин абстрактных состояний и охватывает основные конструкции диаграммы, такие как класс, интерфейс, атрибут, константа, метод, конструктор, отношения наследования и реализации.

Преимуществом предложенного подхода является использование машин абстрактных состояний, а также простейших понятий теории множеств и многоосновных алгебр.

Статья не покрывает часть конструкций диаграммы классов. Работа над построением многопакетных программ, формализацией отношений агрегации и композиции являются предметом дальнейших исследований.

СПИСОК ЛИТЕРАТУРЫ

1. **Буч Г., Рамбо Д., Джекобсон А.** Язык UML. Руководство пользователя. — М.: ДМК, 2000.
2. **UML 1.5** Final Adopted Specification. — OMG, March 2003, http://www.omg.org/uml.
3. **Genova G., Llorens J., Quintana V.** Digging into Use Case Relationships // Lect. Notes Comput. Sci. — 2002. — Vol. 2460. — P. 115–127.
4. **Gogolla M., Henderson-Sellera B.** Analysis of UML Stereotypes within the UML Metamodel // Lect. Notes Comput. Sci. — 2002. — Vol. 2460. — P. 84–99.
5. **Paige R., Ostroff J.** Metamodelling and Conformance Checking with PVS // Lect. Notes Comput. Sci. — 2001. — Vol. 2029. — P. 2–16.
6. **Overgaard G.** Formal Specification of OO Modeling // Lect. Notes Comput. Sci. — 2000. — Vol. 1783. — P. 193–207.
7. **Lellahi K.** Conceptual Data Modeling: An Algebraic Viewpoint // Lect. Notes Comput. Sci. — 2001. — Vol. 2244. — P. 336–348.
8. **Gurevich Y.** Draft of the ASM Guide, May 1997, http://www.eecs.umich.edu/gasm/.

9. **Гуревич Ю.** Последовательные машины абстрактных состояний охватывают последовательные алгоритмы // Системная информатика. Вып. 9 / Пер. П.Ф. Емельянова. — Новосибирск: Изд-во СО РАН, 2004. — С. 7–50.
10. **Lellahi K., Zamulin A**. Object-oriented database as a dynamic system with implicit state // Lect. Notes Comput. Sci. — 2001. — Vol. 2151. — P. 239–259.
11. **Lellahi K., Zamulin A.** Implicit State Approach for Formalization of Sequential Java-like Programs. — Paris, 2002. — (Tech. rep. / LIPN 2002-10, Univ. Paris 13).
12. **Gaudel M.-C., Khoury C., Zamulin A.** Dynamic Systems with Implicit State // Lect. Notes Comput. Sci. — 1999. — Vol. 1577. — P. 114–128.

С. О. Веретнов

ТРАНСЛЯЦИЯ ЯЗЫКА ВЫПОЛНИМЫХ СПЕЦИФИКАЦИЙ РАСПРЕДЕЛЕННЫХ СИСТЕМ SDL В ЯЗЫК ВЫПОЛНИМЫХ СПЕЦИФИКАЦИЙ REAL

ВВЕДЕНИЕ

Последние годы заметно возрастает роль формальных методов, применяемых для разработки распределенных систем, таких как, например, коммуникационные протоколы. Это связано с тем, что для современных распределенных систем усложняется документирование, тестирование и верификация. Для преодоления указанных трудностей используются языки выполнимых спецификаций SDL [1], Estelle, LOTOS, принятые в качестве стандарта международной организацией ITU (International Telecommunication Union). Среди этих языков наиболее активно на практике применяется язык SDL. Верификация выполнимых спецификаций, представленных на языке SDL, заключается в проверке корректности их ключевых свойств и является проблемой современного программирования. Идея перспективного подхода к проблеме верификации SDL-спецификаций состоит в разработке модельных языков, ориентированных на их верификацию, которые были бы комбинированными, т.е. включали подъязыки выполнимых и логических спецификаций.

Примером комбинированного модельного языка спецификаций служит язык REAL, разработанный в лаборатории теоретического программирования Института Систем Информатики СО РАН [2, 3]. Язык REAL базируется на языке SDL и логике ветвящегося времени CTL. Его ядро — язык выполнимых спецификаций Basic-REAL(bREAL), достоинство которого — фиксированная полная структурная операционная семантика выполнимых спецификаций в виде систем переходов. Дальнейшее расширение языка Basic-REAL содержит динамические конструкции и получило название Dynamic-REAL(dREAL).

Целью данной работы является разработка и реализация системы трансляции выразительного подмножества языка SDL в язык выполнимых спецификаций REAL. В качестве языка для входной спецификации взято подмножество языка SDL-88 без использования таких конструкций, как гене-

раторы, сервисы и некоторые другие. В качестве языка выходной спецификации взят язык Dynamic-REAL.

Результаты этой работы были представлены на конференции-конкурсе «Технологии Microsoft в теории и практике программирования» [5].

1. ОБЩАЯ СХЕМА ТРАНСЛЯТОРА

Процесс трансляции содержит две основных стадии.

1. Генерация внутреннего представления, соответствующего тексту входного файла. Во время этого этапа происходит проверка входного текста на предмет того, что он является синтаксически корректной SDL-программой. Результатом является внутреннее представление, максимально приближенное по структуре к bREAL-программе.
2. Генерация bREAL-текста (выходного файла), соответствующего внутреннему представлению. По заданным для всех конструкций языка bREAL-шаблонам происходит построение bREAL-текста.

Результатом работы транслятора является bREAL-текст, соответствующий входной SDL-программе, если она не содержала синтаксических ошибок и представима в языке выполнимых спецификаций bREAL. Также выдаются сообщения об ошибках, показывающих причину, по которой построение выходного файла провести не удалось. Общая схема транслятора дана на рис. 1.

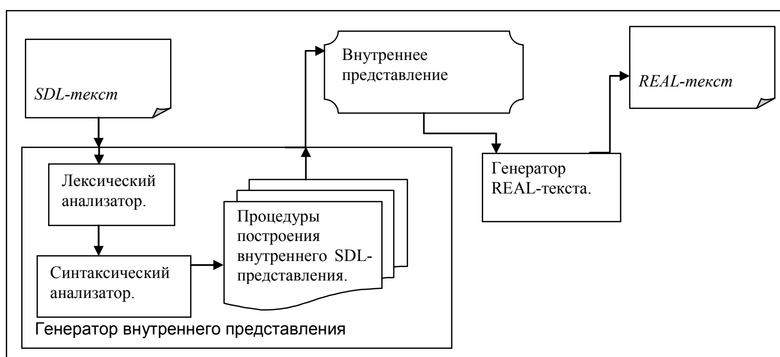


Рис. 1. Схема транслятора из SDL в REAL

1.1. Внутреннее представление входных спецификаций

Текст SDL-программы подается на вход лексическому анализатору, осуществляющему разбивку текста на лексемы. Синтаксический анализатор выделяет синтаксически законченные конструкции и вызывает процедуры генерации внутреннего представления. Затем происходит соединение каналов, создание списков сигналов, с которыми работают процессы, и ряд других действий, которые завершают построение внутреннего представления.

Во время выполнения этого этапа происходит проверка входного текста на его синтаксическую правильность. А также, частично, на его соответствие подмножеству языка SDL, которое может быть переведено в bREAL. В процессе построения внутреннее представление максимально возможно приближается к построению программ на языке bREAL. Во время этого процесса происходит окончательная проверка представимости входного текста в выполнимых спецификациях языка bREAL.

Объект самого высокого уровня — это служебный “глобальный” блок, с точки зрения внутреннего представления он является обычным SDL-блоком и содержит предопределенные объекты с глобальной областью видимости, такими как, например, предопределенные типы данных.

Непосредственным потомком этого блока является *блок-система*, потомками которой являются блоки и далее — процессы. Таким образом, иерархия внутреннего представления практически полностью совпадает с иерархией языка SDL. С точки зрения внутреннего представления процесс — это блок, имеющий граф состояний.

Атрибутами блока являются следующие конструкции.

- Список типов. Типы «Глобального блока» — предопределенные, их определение задается транслятором до начала работы.
- Список переменных. Переменные, определенные константами.
- Список сигналов и список списков сигналов.
- Список каналов и временный список соединений.

Процессы имеют информацию, в которой содержится граф состояний и список условий, если таковые имеются. При трансляции могут использоваться ряд служебных свойств, как например, список сигналов, которые может получать процесс.

2. ПОСТРОЕНИЕ BREAL-ПРЕДСТАВЛЕНИЯ

Блоки/процессы, типы данных и переменные языка SDL имеют прямые аналоги в bREAL, поэтому эта часть процесса трансляции не представляет никакой сложности.

2.1. Каналы

Построение ***каналов*** происходит в два этапа: сначала создаются списки ***каналов*** и ***соединений***, в которых запоминается декларативная информация. Затем, после построения дерева блоков, происходит обработка ***соединений***: проверяется непротиворечивость указанной в них информации и связывания их с конкретными объектами внутреннего представления (блоками/процессами или каналами). Это необходимо, так как SDL допускает использование объектов до их определения.

Семантика каналов в SDL и bREAL имеет ряд существенных отличий: SDL-каналы могут разветвляться или сходиться на границе блоков, в то время как bREAL-каналы присоединяются строго один к одному. Двусторонние каналы преобразуются в два — с множеством сигналов, которые передаются в одном и в противоположном направлении. Возможность разветвления SDL-каналов преодолевается следующим образом: каналы, которые разделяются или соединятся в один, преобразуются таким образом, чтобы образовавшиеся в результате этой операции каналы были односторонними.

Пример 1. *Трансляция двусторонних каналов.*

Пусть в спецификации, написанной на языке SDL, существует канал, описанный следующим образом:

```
SIGNAL
  coin    (/* nominal */ integer),
  station (/* station */ integer),
SIGNALROUTE slot
  FROM Passenger TO Slotmachine WITH coin;
  FROM Slotmachine TO Passenger WITH station;
```

Тогда при трансляции он разобьется на два разнонаправленных канала и его описание будет выглядеть так:

```
INN UNB QUEUE CHN slot
 FOR coin
```

```
 WITH PAR p1 OF integer.
INN UNB QUEUE CHN slot_inv
 FOR station
 WITH PAR p1 OF integer.
   FROM Passenger CHN slot  TO Slotmachine.
 FROM Slotmachine CHN slot_inv  TO Passenger.
```

Создается дополнительный канал slot_inv, противонаправленный каналу slot и способный передавать сигнал station.

2.2. Чтение сигнала

Зафиксируем SDL-состояние. Для каждого оператора **INPUT** в этом состоянии создается bREAL-переход с тем же именем. Сработает то из них, сигнал которого придет первым. Если оператор **INPUT** имеет разрешающее условие, перед состоянием, содержащим оператор **READ**, вставляется bREAL-переход с оператором **WHEN condition**, с нулевым временем ожидания и переходом к чтению сигнала.

Дополнительно, для каждого, не сохраненного и не ожидаемого сигнала (сигнала для которого нет оператора **INPUT**, но который может находиться во входном порту процесса) добавляются альтернативы, читающие этот сигнал с нулевым временем ожидания. Таким образом, если первым сигналом в канале будет один из этих сигналов, он будет отброшен, как этого и требует семантика языка SDL.

Пример 2. *Трансляция SDL-состояния в bREAL-переходы, помеченные одним состоянием.*

Пусть есть SDL-состояние **state1** и процесс, которому оно принадлежит, может принимать сигналы **sig1**, **sig2**.

```
STATE state1;
  INPUT sig1;
  NEXTSTATE state2;
  INPUT sig2;
  NEXTSTATE state3;
ENDSTATE;
```

Тогда, bREAL-код, реализующий это состояние, будет выглядеть следующим образом:

```
TRANSITION state1:
  READ sig1 from channel
  FROM NOW TO INF
JUMP state2.
```

```
TRANSITION state1:
  READ sig2 from channel
  FROM NOW TO INF
JUMP state3.
```

2.3. Тело состояния

Последовательность операторов, образующих тело SDL-состояния, переводится в один или несколько bREAL-переходов по следующим правилам.

1. Каждый SDL-оператор **OUTPUT** переходит в соответствующий ему bREAL-оператор **WRITE**, помещаемый в новый переход.
2. Тело оператора **TASK** (список присваиваний), по возможности, помещается в один bREAL-переход (это невозможно сделать, если выражения содержат операторы экспорта/импорта или обозреваемые переменные). При этом, если текущий bREAL-переход содержит оператор **READ** или **WRITE**, создается новый переход.
3. Если SDL-выражения содержат операторы **EXPORT**, **IMPORT** или **VIEW**, перед bREAL-переходом, использующим это выражение, вставляется группа служебных переходов, реализующих эти конструкции.
4. Если левая часть оператора присваивания является переменной, объявленной в секции **VIEWED** (обозреваемой), после этого выражения вставляются служебные переходы, осуществляющие действия необходимые для отправки нового значения этой переменной адресатам.
5. Для SDL-оператора **DECISION** создается группа альтернативных bREAL-переходов, помеченных одним состоянием.
6. SDL-операторы SET, RESET переходят в bREAL-операторы WRITE.

2. 4. Оператор DECISION

Операторы ветвления языков SDL и bREAL имеют значительные отличия. Кроме того, ограничение на единственность операторов в bREAL-переходе создает дополнительные проблемы при трансляции этих конструкций.

Условный оператор SDL имеет следующий вид:

DECISION условие;
{ {вариант:}+ переход }+
[ELSE: переход]
ENDDECISION;

где

- **условие** — выражение произвольного типа,
- **вариант** — ограничение на множество значений ответа — константа (или множество констант), или, один или несколько интервалов (открытых или закрытых),
- **переход** — последовательность произвольных команд (за исключением оператор **INPUT**) языка SDL.

Условный оператор bREAL имеет вид:

WHEN выражение-условие программа

или

IF выражение-условие THEN программа [ELSE программа] FI

где

- **выражение-условие —** обычное выражение, результат которого имеет логический тип,
- **программа** — последовательность из одного или нескольких операторов bREAL.

Для каждого SDL-варианта создается собственный альтернативный переход — начало ветвления. Оно содержит оператор IF или WHEN с условием, полученным из SDL-условия. Выражение-условие для bREAL-операторов IF или WHEN строится следующим образом.

- Если вариант — константа, то выражение-условие выглядит как *условие = констатнта.*
- Если вариант — открытый интервал, то выражение-условие выглядит как *условие { < | > | <= | >= } граница*.
- Если вариант — закрытый интервал, то выражение-условие выглядит как *(условие <= правая-граница) AND (условие >= левая граница)*.
- Если список вариантов, то выражение-условие выглядит как *выражение-вариант { OR выражение-вариант }+*.
- Выражение-вариант для варианта-ELSE строиться как отрицание дизъюнкции всех вариантов оператора DECISION.

Кроме того создаются состояния и для конца условия (end_decision).

REAL-состояние начала ветвления выглядит так:

```
TRANSITION begin_decision_N
  EXE SKIP
  FROM NOW TO INF
JUMP state_name_T.

TRANSITION state_name_T
  WHEN условие вариант
  FROM NOW TO INF
JUMP state2.
```

Оно срабатывает, только если условие из оператора **WHEN** истинно и осуществляет переход к последовательности состояний реализующих тело варианта. Эти состояния строятся по обычным правилам, описанным выше.

Кроме того, если **DECISION** не содержит варианта **ELSE**, будет добавлено служебное bREAL-состояние с **ELSE**-выражением. Если этого не сделать, процесс попал бы в ситуацию тупика, в случае, когда выражение **условие** не удовлетворяло бы ни одному **варианту**.

Проиллюстрируем трансляцию оператора DECISION на примере.

Пример 3.*Трансляция оператора DECISIO.*

Следующий фрагмент SDL-состояния содержит условие типа integer и 2 варианта:

```
STATE normal;
. . . . .
  DECISION r;
     ( = 0 ): NEXTSTATE normal;
     ( = 1 ): NEXTSTATE restore;
ENDDECISION;
```

Этот фрагмент будут реализовывать следующие bREAL-переходы:

```
TRANSITION begin_decision_1
EXE SKIP
FROM NOW TO INF
JUMP normal_1.

TRANSITION normal_1
WHEN (r=0)EXE SKIP
FROM NOW TO INF
JUMP normal.

TRANSITION normal_1
WHEN (r=1)EXE SKIP
FROM NOW TO INF
JUMP restore.
TRANSITION normal_1
WHEN (NOT ((r=0) OR (r=1)))
EXE SKIP
FROM NOW TO INF
JUMP end_decision_1.

TRANSITION end_decision_1
EXE SKIP
FROM NOW TO INF
JUMP end_of_process.
```

2.5. Обозреваемые переменные. Операторы EXPORT/IMPORT

Язык SDL имеет механизм, позволяющий процессу узнать текущее значение переменной другого процесса. Это делается с помощью оператора **VIEW (имя-переменной),** который заменяется в **выражении,** в котором он использован, значением переменной **имя-переменной** (эта переменная должна быть объявлена специальным образом). Ничего подобного в языке bREAL нет, поэтому эта конструкция реализуется сторонними средствами.

В процесс, который использует оператор **VIEW**, добавляются служебные переменные, по одной для каждой переменной, которые этот процесс собирается обозревать. И между этим процессом и процессом, предоставляющим свои переменные для обозрения, проводиться служебный канал **view-channel** для передачи значений переменных.

В процессе, предоставляющем свои переменные для обозрения, после каждого выражения, в левой части которого стоит обозреваемая переменная, вставляются переходы, помещающие в канал **view-channel** сигнал с новым значением переменной.

В процессе-обозревателе перед переходом, использующим оператор **VIEW**, вставляется следующий код:

```
TRANSITION state_view:
  READ viewvar_ var_name (viewvar_var_name)
FROM view_channel
  FROM NOW TO INF
JUMP state_view.
```

```
TRANSITION state_view:
  WHEN EMP(view_channel)
  EXE SKIP
  FROM NOW TO INF
JUMP state_with_view_expression.
```

А сам оператор **VIEW** заменяется в выражении на служебную переменную **viewvar_ var_name.**

Первое состояние этого фрагмента будет срабатывать до тех пор, пока в канале **view_channel** есть сигналы **viewvar_ var_name** (новые значения этой переменной). Если в канале не окажется требуемого сигнала, это будет означать, что переменная **viewvar_ var_name** содержит текущее значение обозреваемой переменной и сработает вторая альтернатива состояния.

SDL-операторы **EXPORT/IMPORT** также не имеют аналогов в языке bREAL. Их реализация в целом аналогична механизму обозреваемых переменных.

IMPORT заменяется по правилам оператора **VIEW** таким же bREAL-шаблоном. А оператор **EXPORT** переходит в **WRITE**, аналогичный тому, что вставляется после присваиваний, изменяющих значение обозреваемой переменной.

2.6. Таймеры

Механизм таймеров реализуется следующим образом: в блок, которому принадлежит процесс, использующий таймеры, добавляются служебные процессы (по одному на каждый таймер), которые эмулируют работы SDL-таймера. Это делается следующим образом – SDL-оператор **SET** переводится в bREAL-оператор **WRITE** с сигналом **SET**, параметром которого является выражение. Получив этот сигнал, процесс-таймер переходит к альтернативным переходам:

- срабатывает при получении процессом сигнала **RESET**, после чего процесс переход в состояние ожидания;
- при получении нового сигнала **SET** со временем срабатывания, равным полученному при установке выражению, переходит к состоянию, которое отправляет сигнал при срабатывании таймера, после чего процесс переходит в состояние ожидания.

Пример 4.*Служебный процесс-таймер.*

```
PR VAR delay OF integer;

TRANSITION inactive:
  READ set(delay) FROM input;
  FROM NOW TO INF
JUMP active;

TRANSITION inactive:
  READ reset FROM input;
  FROM NOW TO INF
JUMP inactive;

TRANSITION active:
  READ set(delay) FROM input;
  FROM NOW TO INF
JUMP active;

TRANSITION active:
  READ reset FROM input;
  FROM NOW TO INF
JUMP inactive;

TRANSITION active:
  EXE IF (delay <= 0) THEN SKIP ELSE ABRT
  FROM NOW TO INF
JUMP timeout;

TRANSITION active:
  EXE IF (delay > 0) THEN delay = delay - 1;
  SKIP;
  ELSE ABRT
  FROM 1 sec UPTO 1 sec;
JUMP active;

TRANSITION timeout:
  WRITE timeout INTO output;
  FROM NOW TO INF
JUMP inactive;
```

3. ТРАНСЛЯЦИЯ ДИНАМИЧЕСКИХ КОНСТРУКЦИЙ ЯЗЫКА SDL

Благодаря тому что семантика операторов динамического подмножества языка dynamic-REAL не сильно отличается от семантики операторов того же подмножества языка SDL, была предложена следующая логика трансляции.

3.1. Оператор CREATE. Оператор, позволяющий динамически порождать новый экземпляр процесса

Порождение процесаа в SDL осуществляется с помощью оператора CREATE <имя процесса>. В REAL оператор порождения практически не отличается от CREATE PROCESS <имя процесса>. Поэтому трансляция не представляет трудности.

Пример 5. *Трансляция оператора CREATE.*

В SDL:

```
STATE init;
  INPUT frame;
   CREATE Monitor;
NEXTSTATE state1;
```

В dREAL:

```
TRANSITION init
  READ frame FROM s_to_m
  FROM NOW TO INF
 JUMP init_X1.
```

```
TRANSITION init_X1
  CREATE PROCESS Monitor
  FROM NOW TO INF
JUMP state1.
```

3.2. Сигнал KILL. Сигнал, убивающий экземпляр процесса

Несмотря на то что в REAL существует специальный зарезервированный сигнал KILL, в SDL такого сигнала нет и уничтожение процесса происходит стандартными способами языка.

Пример 6. *Уничтожение процесса.*

В SDL:

```
STATE init;
INPUT <signal_name>;
STOP;
```

В dREAL:

```
TRANSITION init
READ frame FROM m_to_m
FROM NOW TO INF
JUMP init_stop.
```

```
TRANSITION init_stop                      FROM NOW TO INF
EXE ABRT                                  JUMP end_of_process.
```

3.3. Идентификаторы экземляров процесса

С развитием языка Dynamic-REAL появилась возможность посылать сигналы определенным экземплярам процесса. Сначала по правилам, приведенным выше, строятся каналы и маршруты. Затем, в теле программ при отправлении сигнала добавляется PId экземпляра процесса.

PId могут быть нескольких видов:

– переменная типа PId, которой заранее было присвоено значение;

– OFFSPRING — PId порожденного процесса;

– PARENT — PId процесса-родителя;

– SELF — PId самого себя;

– SENDER — Pid процесса, от которого прошел последний сигнал.

Например, в SDL:

OUTPUT frame TO OFFSPRING

В REAL:

WRITE frame INTO channel[OFFSPRING]

4. МАКРОКОМАНДЫ

Макросредства языка SDL служат для упрощения описания и улучшения обозримости описания. Они состоят из определения макрокоманд и вызовов макрокоманд. На линейном языке определение макрокоманды состоит из заголовка и тела макрокоманды.

Вызов макрокоманды в теле основной программы осуществляется оператором

MACRO <имя_макрокоманды>[<формальные параметры>];

В языке REAL такие способы описания отсутствуют. Поэтому трансляция макрокоманд реализуется сторонними средствами. Создается копия тела определения макрокоманды, в которой все формальные параметры заменяются на соответствующие лексические единицы, указанные в вызове макрокоманды. Затем из описания системы удаляется вызов макрокоманды и на его место вставляется указанная копия тела макрокоманды.

Пример 7. *Преобразование макрокоманды.*

До преобразования

```
MACRODEFINITION Exam
FPAR alfa, c, s, x;
    BLOCK alfa REFERENCED;
    CHANNEL c FROM x TO alfa WITH s;
    ENDCHANNEL c;
ENDMACRO Exam;
……………………..
BLOCK A REFERENCED;
MACRO Exam(B,C1,S1,A);
BLOCK C REFERENCED;
CHANNEL C2 FROM B TO C WITH S2;
ENDCHANNRL C2;
```

После преобразования

```
……………………..
BLOCK A REFERENCED;
BLOCK B REFERENCED;
CHANNEL C1 FROM A TO B WITH S1;
ENDCHANNRL C1;
BLOCK C REFERENCED;
CHANNEL C2 FROM B TO C WITH S2;
ENDCHANNRL C2;
……………………..
```

5. ГЕНЕРАЦИЯ ВЫХОДНОГО BREAL-ТЕКСТА

Результатом работы этого этапа является REAL-текст, соответствующий входной SDL-программе.

Каждый объект внутреннего REAL-представления имеет функцию, обеспечивающую генерацию текста в соответствии с некоторым шаблоном, заложенным в нее перед началом компиляции. Этот шаблон не зафиксирован и может быть легко изменен. Шаблон представляет собой некоторый текст, включающий в себя ряд спецсимволов, означающих какой-либо элемент внутреннего представления, который может присутствовать в этом объекте. Кроме того, спецсимволами являются команды форматирования текста, такие как перевод строки, структурный отступ и т.д.

Построение REAL-текста начинается вызовом функции MakeRealText для глобального блока (блока самого верхнего уровня), в процессе работы этой функции произойдет рекурсивное развертывание всего внутреннего представления.

ЗАКЛЮЧЕНИЕ

Система трансляции языка SDL в Dynamic-REAL реализована на языке C++. Синтаксический анализатор построен генератором синтаксических анализаторов BISON (GNU, v. 1.28). Система успешно протестирована на

различных примерах, среди которых отметим такие протоколы, как «Касса-Пассажир» [2,3] и кольцевой протокол (Ring protocol) [4].

Транслятор с языка SDL в язык REAL является важной частью системы верификации коммуникационных протоколов, написанных на языке SDL. Другими частями этой системы являются симулятор REAL-спецификаций, а также верификатор REAL-спецификаций. На данном этапе эта система разработана для языка Basic-REAL, а для его расширения — Dynamic-REAL, система находится в стадии разработки.

СПИСОК ЛИТЕРАТУРЫ

1. Карабегов А.В., Тер-Микаэлян Т.М. Введение в язык SDL. — М.: Радио и связь, 1993.
2. Непомнящий В.А., Шилов Н.В., Бодин Е.В. REAL: язык для спецификации и верификации систем реального времени // Системная информатика. Вып. 7. — Новосибирск: Наука, 2000. — С. 174–224.
3. Nepomniaschy V.A., Shilov N.V., Bodin E.V., Kozura V.E. Basic-REAL: integrated approach for design, specification and verification of distributed systems // Lect. Notes Comput. Sci. — 2002. — Vol. 2335. — P. 69–88.
4. Cohen R., Segall A., An efficient reliable ring protocol // IEEE Transactions on Communications. — 1991. — Vol. 39, N 11. — P. 1616–1624.
5. Веретнов С.О. Разработка и реализация транслятора с языка спецификаций SDL в язык выполнимых спецификаций REAL // Конференция-конкурс «Технологии Microsoft в теории и практике программирования». — Новосибирск, 2006. — С. 4–5.

Н.К. Вольхина

АВТОМАТИЧЕСКОЕ ВОССТАНОВЛЕНИЕ БИЗНЕС-ЛОГИКИ ПРОГРАММ

Сопровождение программ, изменяющихся в течение длительного периода, — достаточно трудоемкий процесс, даже если им занимается автор кода и программа написана на одном из современных языков. Очевидно, для старых, «унаследованных» (legacy), систем поддержка еще более усложняется. Тем не менее, многие крупные организации сегодня продолжают использовать приложения, написанные на языках типа Cobol, PL/I, Natural и др. Сопровождение таких систем требует существенных затрат, поскольку для каждого даже небольшого изменения кода необходим тщательный предварительный анализ. Нетрудно заметить, что восстановление бизнес-логики, заложенной в приложение при его разработке, является более эффективным, чем многократный частичный анализ кода. Этот процесс также является затратным, но он проводится один раз, и его результаты существенно упрощают дальнейшее сопровождение, а так же могут быть использованы для документирования и реинжиниринга системы.

Практика показывает, что восстановление бизнес-логики не может быть полностью автоматическим и требует непосредственного человеческого участия. Для реальных приложений это сложный длительный процесс, что делает актуальной частичную его автоматизацию при помощи специальных методов и инструментальных средств. Один из таких методов, называемый далее AutoDetect, описывается в данной статье. Основываясь на информационном графе программы, AutoDetect строит бизнес-правила, которые далее редактируются человеком. Метод реализован в рамках компоненты Business Rule Manager системы Modernization Workbench [12], предназначенной для анализа и преобразования больших приложений.

Под *бизнес-правилом* мы будем понимать именованную сущность, соответствующую фрагменту кода и документирующую его. Предлагаемая функциональность сопоставляет бизнес-правила операторам программы, указывая условия их исполнения, входные и выходные данные. Для выбранной переменной в коде AutoDetect позволяет восстановить последовательность операторов, вычисляющих данную переменную. Здесь и далее *переменной* называется каждое отдельное вхождение программных переменных. В дальнейшем будем называть автоматическое нахождение бизнес-правил, т. е. работу AutoDetect, *автодетектированием*.

В первой части статьи показывается, что AutoDetect получает на вход, и вводится пример, на котором далее демонстрируется процесс автодетектирования. Во второй части описаны построение и преобразования операторного графа — основной структуры, с которой работает AutoDetect. Третья часть посвящена собственно генерации групп бизнес-правил. В четвертой части приводится обзор аналогичных разработок. Дальнейшие направления деятельности приведены в заключении.

1. ВХОДНЫЕ ДАННЫЕ ПРОЦЕССА АВТОДЕТЕКТИРОВАНИЯ

Поскольку AutoDetect является одной из множества функциональностей большой системы, он тесно связан с другими средствами анализа. Так при запуске процесса автодетектирования в качестве входных данных выступает использующийся в системе *специальный информационный граф*. Как и стандартный информационный граф [2, 3], он строится на основе анализа потоков данных [1, 2], ведущих к указанной переменной. Особенность специального графа состоит в том, что он содержит дополнительную информацию об условных зависимостях.

Вершинами графа являются переменные, необходимые для вычисления выбранной, т.е. такие переменные, значения которых либо 1) *явно используются* в вычислениях, либо 2) *контролируют* их. Для каждой переменной система может определить охватывающую конструкцию языка: оператор или условие. В соответствии с этим переменные первого типа можно назвать *операторными*, а переменные второго типа — *условными*. Соответственно граф имеет два типа ребер: информационные и условные. *Информационные* ребра соответствуют информационным зависимостям в программе: ребро из A в B показывает, что переменная B зависит от переменной A, т.е. для вычисления значения B *используется значение* A. *Условное* ребро из A в B означает, что вычисление значения переменной B *зависит от значения* переменной A. В этом случае A является контролирующей переменной для вычисления переменной B.

Для большей ясности результата вычисление условных переменных не анализируется. При необходимости можно запустить AutoDetect для каждой такой переменной непосредственно.

В качестве примера для демонстрации процесса автодетектирования рассмотрим фрагмент программы на языке Cobol (рис. 1), вычисляющий скидку (переменная DISCOUNT в последней строке) по некоторым правилам. Легко заметить, что не все операторы приведенного фрагмента ис-

пользуются для ее вычисления. На рис. 1 неиспользуемые операторы отмечены курсивом. В частности, считаем таковыми операторы MOVE 1 TO I и ADD 1 TO I, которые нужны только для вычисления переменной из условия.

Операторы PERFORM и GO TO участвуют в вычислениях неявно, осуществляя лишь передачу управления. Бизнес-правила из таких операторов не создаются.

```
INIT.
    MOVE 0 TO DISCOUNT-FACTOR1 DISCOUNT-FACTOR2.
    MOVE 1 TO I.
    IF UPDATE-NEEDED = "TRUE" THEN
        IF CURDAY > 5 THEN
            COMPUTE DISCOUNT-FACTOR2 = PRICE-0 - 50
            MOVE PRICE-0 TO PRICE
            MOVE 0 TO TAX
        ELSE
            COMPUTE PRICE = PRICE-0 + 10
            MOVE 0.6 TO TAX
        END-IF
    END-IF.

FACTOR-CALC.
    IF I > MAX THEN
        GO TO MAIN-CALC.
    COMPUTE RES = DISCOUNT-FACTOR1 + CONST.
    ADD CONSTS TO RES.
    COMPUTE DISCOUNT-FACTOR1 = RES * 0.25.
    ADD 1 TO I.
    PERFORM FACTOR-CALC.

MAIN-CALC.
    ADD DISCOUNT-FACTOR2 TO RES.
    COMPUTE DISCOUNT-FACTOR2 = 0.5 * DISCOUNT-FACTOR2.
    COMPUTE TAX = TAX * DISCOUNT-FACTOR2 + RES.
    COMPUTE DISCOUNT = DISCOUNT-FACTOR1 * PRICE + DISCOUNT-FACTOR2.
```

Рис. 1. Исходный фрагмент программы на языке Cobol

Информационный граф для переменной DISCOUNT приведен на рис. 2. Здесь белые вершины соответствуют операторным переменным, серые — условным. Ребро, помеченное знаком «+», показывает, что имеется зависимость от условия, в которое входит данная переменная. Знак «–» означает зависимость от отрицания условия. Например, в случае оператора IF «+»-ребро отвечает ветке THEN, а «–»-ребро — ветке ELSE.

Рис. 2. Информационный граф, приходящий на вход процесса автодетектирования

2. ОПЕРАТОРНЫЙ ГРАФ И ЕГО ПРЕОБРАЗОВАНИЯ

Бизнес-правила создаются из операторов, поэтому, имея граф переменных, процесс автодетектирования на его основе строит операторный граф. Это несложно сделать, поскольку для переменных известны охватывающие конструкции языка — операторы или условия.

После построения операторного графа все дальнейшие преобразования, необходимые для формирования последовательности бизнес-правил, работают с ним. Большой набор алгоритмов работы с графами можно найти в [3, 4].

2.1. Построение графа

Рассмотрим процесс построения графа более подробно.

Вершины исходного графа, соответствующие переменным одного оператора или условия, объединяются в одну операторную вершину. Одному условию могут соответствовать две вершины: для собственно условия и для его отрицания. Если в исходном графе только один тип ребер («+» или «–»)

для данного условия, то в операторном графе создается только одна вершина (без отрицания или с отрицанием соответственно). В результате каждая вершина операторного графа обозначает либо оператор, либо условие, либо отрицание условия. Ребра также поднимаются на операторный уровень: если в исходном графе есть ребра, соединяющие переменные из двух операторов, то создается ребро между операторными вершинами.

Описанное преобразование позволяет получить граф, который содержит информацию обо всех операторах, используемых для вычисления выбранной переменной, зависимостях по данным между операторами и зависимостях от условий.

Заметим, что при переходе от вершин-переменных к вершинам-операторам в последние записывается вся информация о переменных оператора как о его входных и выходных данных.

На рис. 3 показано, как из вершин исходного графа формируются операторные вершины: переменные одного оператора выделены прямоугольными рамками.

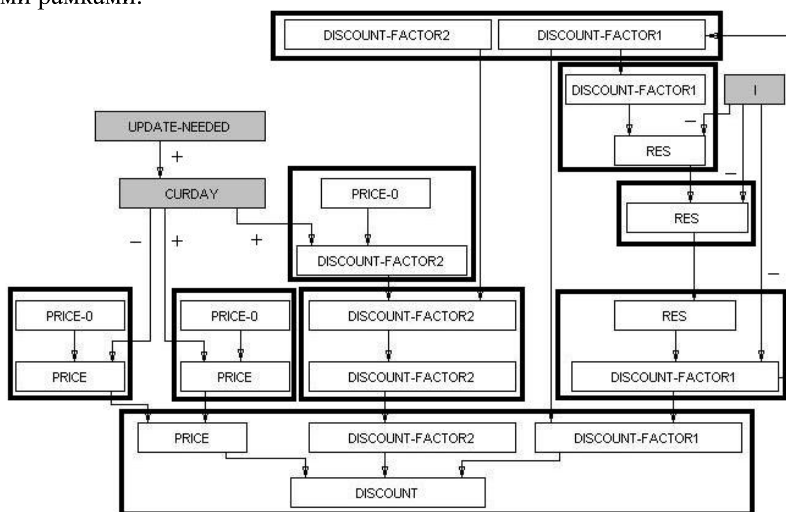


Рис. 3. Группирование вершин, соответствующих переменным одного оператора

Рис 4. демонстрирует результирующий операторный граф, где белые вершины обозначают операторы, серые вершины — условия, черные ребра — зависимости по данным, серые ребра — условные зависимости.

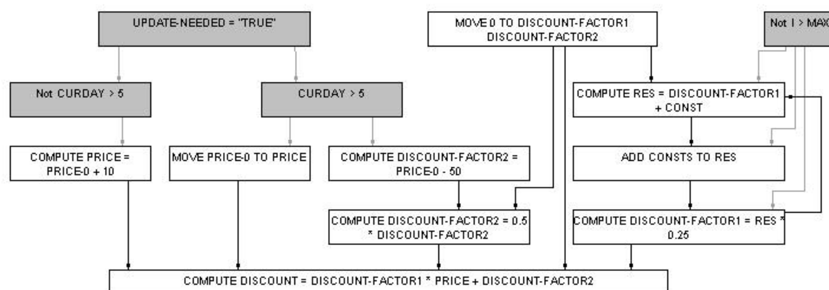


Рис. 4. Операторный граф для информационного графа, приведенного на рис. 2

Далее рассмотрим преобразования операторного графа, которые применяет AutoDetect для формирования последовательности бизнес-правил.

2.2. Разрезание циклов

Построенный операторный граф имеет единственный выход, соответствующий оператору с интересующей переменной, и несколько входов. Граф может содержать циклы, поскольку ограничения на исходную программу не накладывались. Однако последовательность бизнес-правил предполагает, что каждый оператор, обозначенный бизнес-правилом, выполнится ровно один раз. Таким образом, AutoDetect любой цикл программы заменяет одной его итерацией. Это ограничение в дальнейшем планируется устранить.

В терминах операторного графа замена цикла одной итерацией означает удаление ребра, замыкающего цикл. Обнаружение таких ребер производится обходом графа в глубину в обратном направлении ребер, начиная с единственного выхода. Для обнаружения неориентированных циклов достаточно помечать вершины в процессе обработки, тогда попадание в помеченную вершину означает замыкание цикла. В нашем случае необходимо разрезать только ориентированные циклы, поэтому, кроме механизма меток, используется вспомогательный стек для хранения пройденных вершин. Стек позволяет обнаруживать циклы, а механизм пометок отменяет повторный поиск в глубину от вершин, из которых выходит более одного ребра.

Для рассматриваемого примера в операторном графе присутствует один цикл, замыкающее его ребро показано серым цветом на рис. 5.

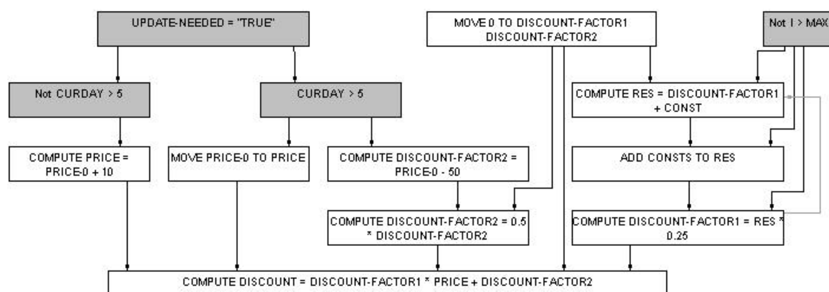


Рис. 5. Операторный граф с обнаруженным циклом

2.3. Топологическая сортировка

К полученному после предыдущего преобразования ациклическому графу применяется топологическая сортировка [5]. При этом, с одной стороны, определяется порядок операторов в последовательности бизнес-правил, а с другой — происходит разбиение графа на уровни, необходимое для последующего корректного определения условий.

На рис. 6 показан граф демонстрационного примера. Вершины одного уровня расположены на одной горизонтали, рядом с каждой операторной вершиной указан порядковый номер, назначенный ей при топологической сортировке.

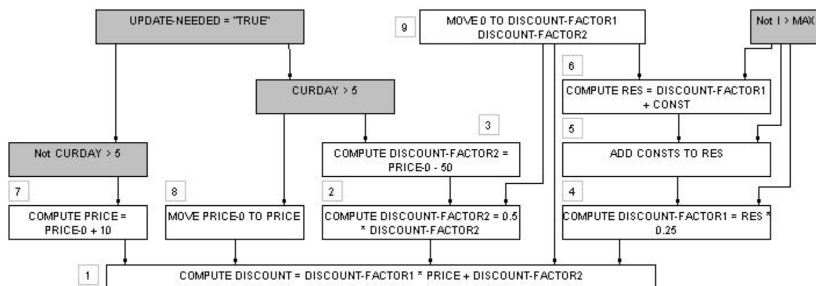


Рис. 6. Операторный граф после применения топологической сортировки

2.4. Определение условий исполнения операторов

Условия исполнения операторов приписываются к бизнес-правилам в качестве дополнительной информации. Поэтому следующим шагом является распространение информации с условных вершин на операторные. При проходе через условную вершину в нее собираются условия с входящих в нее ребер и вместе с условием, хранящимся в вершине, передаются на все выходящие ребра. Для операторных вершин собираются все условия с входящих условных ребер. Для того чтобы к моменту обработки вершины на входящих в нее ребрах уже была записана вся необходимая информация об условиях, граф обходится по уровням, начиная с самого верхнего.

Заметим, что к операторной вершине приписывается набор условий, а не логическое выражение, составленное из них, что является ограничением текущей реализации и предметом дальнейших исследований.

Для операторного графа, приведенного на рис. 6, описанное преобразование распространения условий добавит информацию к операторным вершинам согласно следующей таблице.

Таблица

Условия исполнения операторов

Оператор	Условия исполнения	
COMPUTE RES = DISCOUNT-FACTOR1 + CONST	NOT I > MAX	
ADD CONSTS TO RES	NOT I > MAX	
COMPUTE DISCOUNT-FACTOR2 = PRICE-0 - 50	UPDATE-NEEDED = "TRUE"	CURDAY > 5
COMPUTE DISCOUNT-FACTOR1 = RES * 0.25	NOT I > MAX	
MOVE PRICE-0 TO PRICE	UPDATE-NEEDED = "TRUE"	CURDAY > 5
COMPUTE PRICE = PRICE-0 + 10	UPDATE-NEEDED = "TRUE"	NOT CURDAY > 5

3. ВЫДЕЛЕНИЕ ПОДПОСЛЕДОВАТЕЛЬНОСТЕЙ ИЗ ПОСЛЕДОВАТЕЛЬНОСТИ БИЗНЕС-ПРАВИЛ

После проведения описанных выше преобразований операторного графа мы имеем всю информацию, необходимую для формирования последовательности бизнес-правил:

- порядок операторов в последовательности, соответствующий обратному порядку вершин, заданному при топологической сортировке;
- входные и выходные переменные каждого оператора, полученные при переходе от исходного информационного графа к операторному графу;
- условия исполнения операторов.

В реальных приложениях количество бизнес-правил исчисляется сотнями, и демонстрировать их в виде линейной последовательности неудобно для восприятия. Поэтому предлагается группировать бизнес-правила по некоторым признакам и в исходной последовательности заменять их специальными бизнес-правилами, ссылочными на соответствующие группы. Такие группы бизнес-правил являются аналогами процедур в языках программирования, поэтому можно их условно назвать *бизнес-процедурами*. Ссылочные бизнес-правила служат аналогами вызовов процедур.

Возможны различные критерии выделения бизнес-правил в бизнес-процедуры. Один из вариантов — заменять процедурой одинаковые подпоследовательности, встречающиеся несколько раз внутри исходной, с целью устранения повторов. Другая возможность — группировать бизнес-правила по принадлежности к программе для случая межпрограммных вызовов. Сравнительный анализ всевозможных критериев, а также выяснение возможности их последовательного применения — одно из направлений дальнейших исследований.

В текущей реализации AutoDetect группирует бизнес-правила по условиям исполнения с целью минимизировать повторение одного и того же условия. Подробное описание и обоснование корректности алгоритма выходит за рамки данной статьи. Приведем лишь общую идею:

- условия упорядочиваются по числу повторений в различных бизнес-правилах;
- для каждого условия, начиная с самого частого, выбираются все зависящие от него бизнес-правила и группируются в бизнес-процедуру;
- процесс применяется рекурсивно.

Для рассматриваемого примера вычисления скидки последовательность бизнес-правил приведена на рис. 7, результат группирования по условиям — на рис. 8. Группы обозначены римскими цифрами. Ссылочные бизнес-правила имеют вид «CALL номер группы».

Легко заметить, что полученная программа на языке бизнес-правил эквивалентна исходной программе (с точностью до упомянутых ранее ограничений автодетектирования), однако является более структурированной.

1. **MOVE 0 TO DISCOUNT-FACTOR1 DISCOUNT-FACTOR2**
   Выход: DISCOUNT-FACTOR1, DISCOUNT-FACTOR2
2. *При условиях:* UPDATE-NEEDED = "TRUE", CURDAY > 5
   MOVE PRICE-0 TO PRICE
   Вход: PRICE-0
   Выход: PRICE
3. *При условиях:* UPDATE-NEEDED = "TRUE", NOT CURDAY > 5
   COMPUTE PRICE = PRICE-0 + 10
   Вход: PRICE-0
   Выход: PRICE
4. *При условиях:* NOT I > MAX
   COMPUTE RES = DISCOUNT-FACTOR1 + CONST
   Вход: DISCOUNT-FACTOR1
   Выход: RES
5. *При условиях:* NOT I > MAX
   ADD CONSTS TO RES
   Вход/Выход: RES
6. *При условиях:* NOT I > MAX
   COMPUTE DISCOUNT-FACTOR1 = RES * 0.25
   Вход: RES
   Выход: DISCOUNT-FACTOR1
7. *При условиях:* UPDATE-NEEDED = "TRUE", CURDAY > 5
   COMPUTE DISCOUNT-FACTOR2 = PRICE-0 - 50
   Вход: PRICE-0
   Выход: DISCOUNT-FACTOR2
8. **COMPUTE DISCOUNT-FACTOR2 = 0.5 * DISCOUNT-FACTOR2**
   Вход/Выход: DISCOUNT-FACTOR2
9. **COMPUTE DISCOUNT = DISCOUNT-FACTOR1 * PRICE + DISCOUNT-FACTOR2**
   Вход: DISCOUNT-FACTOR1, PRICE, DISCOUNT-FACTOR2
   Выход: DISCOUNT

Рис. 7. Последовательность бизнес-правил

I. 1) **MOVE 0 TO DISCOUNT-FACTOR1 DISCOUNT-FACTOR2**
2) *При условиях:* NOT I > MAX
CALL II.
3) *При условиях:* UPDATE-NEEDED = "TRUE"
CALL III.
4) **COMPUTE DISCOUNT-FACTOR2 = 0.5 * DISCOUNT-FACTOR2**
5) **COMPUTE DISCOUNT = DISCOUNT-FACTOR1 * PRICE + DISCOUNT-FACTOR2**

II. 1) **COMPUTE RES = DISCOUNT-FACTOR1 + CONST**
2) **ADD CONSTS TO RES**
3) **COMPUTE DISCOUNT-FACTOR1 = RES * 0.25**

III. 1) *При условиях:* NOT CURDAY > 5
COMPUTE PRICE = PRICE-0 + 10
2) *При условиях:* CURDAY > 5
CALL IV.

IV. 1) **MOVE PRICE-0 TO PRICE**
2) **COMPUTE DISCOUNT-FACTOR2 = PRICE-0 - 50**

Рис. 8. Последовательность бизнес-правил с выделенными подпоследовательностями

4. МЕСТО AUTODETECT В РЯДЕ АНАЛОГИЧНЫХ РАЗРАБОТОК

На сегодняшний день существует множество информационных систем, работающих с бизнес-правилами, созданы специальные группы, которые занимаются исследованием бизнес-правил, разрабатываются различные стандарты [6, 10]. Однако большинство систем предлагают возможности для создания бизнес-правил пользователем на основе его знания предметных областей и преобразования этих правил в формат, который в дальнейшем может использоваться в программной реализации. В разработках такого типа построение бизнес-правил происходит на начальном этапе разработки приложений. Другое направление составляют системы, создающие бизнес-правила на основе уже готовых приложений, извлекая бизнес-логику из программного кода (Business Rule Mining). К этому направлению

относится система Modernization Workbench, в рамках которой разрабатывался AutoDetect.

Отметим, что задача восстановления бизнес-правил по коду наиболее востребована для бизнес-приложений, написанных на старых языках программирования. В связи с историческими особенностями развития информационных систем, и в частности бизнес-приложений, в нашей стране нет больших объемов такого кода, требующего поддержки или преобразования, поэтому практически нет подобных разработок.

Различные решения рассматриваемой задачи могут отличаться конечной целью извлечения бизнес-логики и некоторыми особенностями обрабатываемых приложений. Например, существует разработка, ориентированная на приложения с трехуровневой архитектурой: пользовательский интерфейс — бизнес-логика — взаимодействие с базами данных [8].

Две основные конечные цели построения бизнес-правил — это сопровождение приложения и перенос приложения на современный объектно-ориентированный язык. В зависимости от выбора цели отличаются и способы построения бизнес-правил. В первом случае необходима детализированная документация приложения и точная привязка бизнес-правил к исходному коду. Во втором случае допускается первоначальная обработка кода, приведение его к виду, более удобному для объектно-ориентированного подхода. При этом детали текущей реализации могут быть уже неинтересны. Примеры решений второго типа описаны в работах [7, 13].

Описанный в статье AutoDetect относится к первому типу и помогает создать детализированную документацию программы, которая в дальнейшем используется для сопровождения. Другие примеры реализаций, ориентированных на составление детализованной документации или требующих точную привязку к коду, описаны в работах [9, 11]. Более подробному сравнению нашей реализации с другими, но преследующими те же цели, будет посвящена отдельная статья.

5. ЗАКЛЮЧЕНИЕ

В данной статье приведено ознакомительное описание функциональности AutoDetect системы Modernization Workbench, позволяющей по выбранной переменной восстанавливать процесс ее вычисления. Результатом автодетектирования является последовательность бизнес-правил, соответствующих операторам исходной программы, некоторым образом разбитая

на группы. Новая программа на языке бизнес-правил эквивалентна исходной. При этом пользователь имеет возможность задать бизнес-имена входных и выходных переменных, а также изменить имена бизнес-правил, не нарушая их связи с операторами в коде.

Таким образом, AutoDetect является вспомогательным средством как для восстановления бизнес-логики приложения, так и для его документирования.

Текущая реализация имеет некоторые ограничения, устранение которых является одним из направлений дальнейшей работы. Кроме того, планируется продолжить исследование различных способов выделения подпоследовательностей из результирующей последовательности бизнес-правил.

СПИСОК ЛИТЕРАТУРЫ

1. **Ахо А., Сети Р., Ульман Д.** Компиляторы: принципы, технологии и инструменты. — М.: Издательский дом «Вильямс», 2002.
2. **Касьянов В. Н.** Оптимизирующие преобразования программ. — М.: Наука, 1988.
3. **Касьянов В. Н., Евстигнеев В. А.** Графы в программировании: обработка, визуализация и применение. — СПб.: BHV-Санкт-Петербург, 2003.
4. **Кристофидес Н.** Теория графов. Алгоритмический подход. — М.: Мир, 1978.
5. **Свами М., Тхуласираман К.** Графы, сети и алгоритмы. — М.: Мир, 1984.
6. **Business** Rules Group. — http://www.businessrulesgroup.org
7. **Enterprise** Evolution: Modernization. The Transformation Process. — www.ecubesystems.com/ marketing/FAQ%20Compaign/ Enterprise%20Evolution%20-%20The%20Transformation%20Process .pdf
8. **Hung M., Zou Y.** Extracting Business Policies and Business Data from the Three-Tier Architecture Systems // Proc. Internat. Workshop on Reverse Engineering To Requirements, Pittsburgh, Pennsylvania, USA, November 2005. — http://www.cs.toronto.edu/km/retr/camera/hung05retr.pdf
9. **Legacy** IT Applications & Compliance with Sarbanes-Oxley. — www.tringroup.com/images /TMGiSOXWP.pdf
10. **Object** Management Group. — http://www.omg.com
11. **Pocock J.** Business rules Mining: Application Support, Enhancement and Forward Engineering. — 2004. — http://www.transoft.com/products/brm.htm
12. **Relativity** Technologies — Enterprise Application Modernization. — http://www.relativity.com
13. **Sneed H.** Extracting Business Logic from Existing COBOL Programs as a Basis for Redevelopment // Proc. 9th Internat. Workshop on Program Comprehension. — IEEE Computer Society, 2001. — P. 167–175.

Н. С. Грибовская

ОТКРЫТЫЕ МОРФИЗМЫ И ВРЕМЕННАЯ ТЕСТОВАЯ ЭКВИВАЛЕНТНОСТЬ ДЛЯ ВРЕМЕННЫХ АВТОМАТНЫХ МОДЕЛЕЙ

1. ВВЕДЕНИЕ

В последние годы методы теории категорий активно используются для описания и изучения параллельных систем и процессов. Впервые для этих целей теория категорий и ее методы были применены в начале 90-х годов с целью сравнения различных моделей. Однако позже оказалось, что теоретико-категорный подход позволяет также определять и поведенческие эквивалентности между моделями, без которых теория параллелизма просто немыслима.

На сегодняшний день из литературы известно много различных эквивалентностей, взаимосвязи между которыми хорошо изучены. Среди наиболее популярных подходов определения эквивалентностей стоит отметить три подхода: трассовый, тестовый и бисимуляционный. Первый из этих подходов наиболее простой и естественный. Трассовые эквивалентности формулируются в терминах равенства языков моделируемых систем. Такой подход удобен для анализа последовательных систем, однако он не сохраняет информацию о недетерминированном выборе. При тестовом подходе поведение системы исследуется посредством набора тестов. Две системы считаются тестово эквивалентными тогда и только тогда, когда они могут или должны проходить один и тот же набор тестов. Такое определение привело к появлению математической теории, объединяющей естественным образом эквивалентности и предпорядки. При бисимуляционном подходе две системы считаются эквивалентными, если наблюдатель не может обнаружить различий в их поведении с учетом точек недетерминированного выбора. Бисимуляционные эквивалентности являются наиболее сильными, в том смысле, что две бисимуляционно эквивалентные системы автоматически являются трассово и тестово эквивалентыми.

Первыми, кто попробовал использовать методы теории категорий при исследовании эквивалентностей, были Винскель, Нильсен и Джояль [7]. Они предложили рассматривать эквивалентности в терминах существования конструкции открытых морфизмов и показали применимость этого подхода на примере бисимуляционной эквивалентности Милнера [7]. В

дальнейшем предложенная схема стала стандартом и с ее помощью были охарактеризованы и другие эквивалентности (трассовая, слабая и сильная бисимуляционная, бисимуляционная с сохранением истории и т.д.) [8].

В настоящее время резко возрос интерес к разработке и исследованию распределенных систем, функционирующих в режиме реального времени. В результате появились временные варианты различных эквивалентностей, что позволило исследовать временные аспекты поведения систем. Теория категорий и здесь нашла свое применение. Так, например, в статье [5] методами теории категорий была решена проблема разрешимости временной бисимуляционной эквивалентности для временных автоматных моделей, а в статье [9] были получены теоретико-категорные характеризации для различных эквивалентностей в контексте временных моделей с семантикой истинного параллелизма.

Цель данной работы состоит в разработке теоретико-категорной основы для построения и изучения временного варианта тестовой эквивалентности в контексте автоматных моделей реального времени — временных систем переходов.

Статья организована следующим образом. Во втором разделе описывается модель временных систем переходов и приводится ряд обозначений. В разд. 3 строится категория временных систем переходов $CTTS_{test}$ и рассматриваются различные свойства этой категории. Разд. 4 посвящен основным понятиям, связанным с открытым морфизмом. Здесь выделяется подкатегория наблюдений P_{test} в категории $CTTS_{test}$, определяется понятие открытого морфизма и доказывается его критерий. В разд. 5 приводится теоретико-категорная характеризация временной тестовой эквивалентности. Заключение можно найти в разд. 6.

2. МОДЕЛЬ ВРЕМЕННЫХ СИСТЕМ ПЕРЕХОДОВ

В данном разделе описывается модель временных систем переходов и приводится ряд полезных определений и обозначений. Исследуемые временные системы представляют собой обычные временные автоматы без множества поглощающих состояний и условий принятия. Алур и Дилл иногда называют эти модели временными таблицами переходов [3].

Определение 1. Временная система переходов $\mathfrak{T}$ *— это набор (S, Σ,* s_0*, X, T), где*

- *S — множество состояний, а* s_0 *— начальное состояние,*

- *Σ — конечный алфавит действий,*
- *X — множество временных переменных,*
- *T — множество переходов таких, что $T \subseteq S \times \Sigma \times \Delta \times 2^X \times S$.*

Здесь Δ — временная конструкция, построенная по правилу:

$$\Delta ::= c \# x \mid x{+}c \# y \mid \Delta \wedge \Delta,$$

где $\# \in \{\leq, <, \geq, >\}$, c — положительная вещественная постоянная, а x, y — временные переменные. Переход $(s, \sigma, \delta, \lambda, s')$ будем обозначать как $s \xrightarrow{\sigma,\delta,\lambda} s'$.

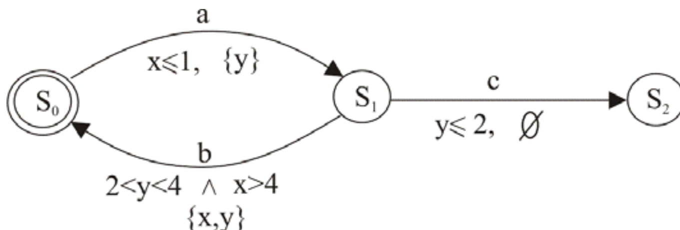


Рис. 1. Временная система переходов $\Im_1$

Пример 1. На рис. 1 изображен пример временной системы переходов $\Im_1$. Для этой системы множество состояний S_1 состоит из трех состояний s_0, s_1 и s_2, что графически изображается кругами, причем начальное состояние s_0 обозначается двойным кругом. Кроме того, алфавит действий Σ_1 содержит три действия: ***a, b*** и ***c***, а множество временных переменных X_1 состоит из двух переменных ***x*** и ***y*.** Переходы между состояниями графически изображаются стрелками. При этом каждая стрелка помечается соответствующим действием, временной конструкцией и множеством временных переменных.

Чтобы объяснить поведение временной системы переходов приведем ряд полезных понятий.

- Множество вещественных неотрицательных чисел будем обозначать как $\boldsymbol{R}_+$.
- Временным словом над алфавитом Σ называется конечная последовательность пар $\alpha = (\sigma_1,\tau_1)(\sigma_2,\tau_2)\ldots(\sigma_n,\tau_n)$, где для любого $0 \leq i \leq n$ верно, что $\sigma_i \in \Sigma$, $\tau_i \in \boldsymbol{R}_+$ и, кроме того, $\tau_i < \tau_{i+1}$.
- Временной функцией прогресса называется функция $\nu : X \to \boldsymbol{R}_+$, которая каждой временной переменной системы сопоставляет кон-

кретный момент времени. Определим временную функцию прогресса следующим образом: $(\nu+c)(x):=\nu(x)+c$ для любой временной переменной x. Кроме того, если λ — множество временных переменных, то $\nu[\lambda \to 0](\mathrm{x}) := \begin{cases} 0, & \text{если } \mathrm{x} \in \lambda, \\ \nu(\mathrm{x}), & \text{иначе.} \end{cases}$

- Будем говорить, что временная конструкция δ выполнена для временной функции прогресса ν тогда и только тогда, когда выражение $\delta[\nu(x)/x]$ истинно. Здесь запись $\delta[y/x]$ означает синтаксическую замену переменной x на переменную y в конструкции δ. Временная конструкция δ определяет подмножество в множестве $(\boldsymbol{R}_+)^m$ (m — число временных переменных в множестве X). Будем называть это подмножество δ-подмножеством и обозначать $||\delta||_X$. Временная функция прогресса ν определяет точку в множестве $(\boldsymbol{R}_+)^m$ (обозначается $||\nu||_X$). Таким образом, временная конструкция δ выполнена для временной функции прогресса ν тогда и только тогда, когда $||\nu||_X \in ||\delta||_X$.

Переходим к рассмотрению поведения временной системы переходов.

Определение 2. *Пусть $\mathfrak{I} = (S, \Sigma, s_0, X, T)$ — временная система переходов. Тогда конфигурацией $\mathfrak{I}$ называется пара $\langle s, \nu \rangle$, где s — состояние, а ν — временная функция прогресса. Конфигурация $C_0(\mathfrak{I}) = \langle s_0, \nu_0 \rangle$, где ν_0 — нулевая постоянная функция, называется начальной конфигурацией. Множество всех конфигураций временной системы переходов будем обозначать как* ***Conf****$(\mathfrak{I})$.*

Будем говорить, что во временной системе переходов $\mathfrak{I}$ существует последовательность выполнения

$$C_0(\mathfrak{I}) \xrightarrow{\sigma_1,\tau_1} \langle s_1, \nu_1 \rangle \xrightarrow{\sigma_2,\tau_2} \ldots \xrightarrow{\sigma_n,\tau_n} \langle s_n, \nu_n \rangle$$

тогда и только тогда, когда для любого $i > 0$ существует переход $s_{i-1} \xrightarrow{\sigma_i,\delta_i,\lambda_i} s_i$ такой, что

$$||\nu_{i-1} + (\tau_i - \tau_{i-1})||_X \in ||\delta_i||_X \text{ и } \nu_i = (\nu_{i-1} + (\tau_i - \tau_{i-1}))[\lambda_i \to 0].$$

Здесь τ_0 равно 0. Эта последовательность выполнения порождает временное слово $(\sigma_1,\tau_1)(\sigma_2,\tau_2)(\sigma_3,\tau_3)\ldots(\sigma_n,\tau_n)$.

Теперь можно привести определение языка временной системы переходов.

Определение 3. Языком временной системы переходов $\Im$ *называется множество* $L(\Im) = \{\ (\sigma_1, \tau_1)(\sigma_2, \tau_2)...(\sigma_n, \tau_n)$ *в* $\Im$ *существует последовательность выполнения вида*

$$\langle s_0, \nu_0 \rangle \xrightarrow{\sigma_1, \tau_1} \langle s_1, \nu_1 \rangle \xrightarrow{\sigma_2, \tau_2} ... \xrightarrow{\sigma_n, \tau_n} \langle s_n, \nu_n \rangle \}.$$

Пример 2. Языком временной системы переходов $\Im_1$, изображенной на рис. 1, является множество $L(\Im_1) = \{(a,d_1)(c,d_2),\ (a,t_1)(b,t_2)\ldots(a,t_{2k-1})(b,t_{2k})(a,t_{2k+1})(c,t_{2k+2})\ |\ 0 < d_1 \le 1,\ (d_2 - d_1) \le 2,\ k \ge 1,\ 1 \le i \le k,\ 0 < (t_{2i-1} - t_{2i-2}) \le 1,\ 2 < (t_{2i} - t_{2i-1}) < 4,\ (t_{2i} - t_{2i-2}) > 4,\ t_0 = 0,\ 0 < (t_{2k+1} - t_{2k}) \le 1,\ (t_{2k+2} - t_{2k+1}) \le 2\}$.

В заключение этого раздела введем ряд полезных понятий и обозначений, необходимых для определения морфизма между двумя временными системами переходов.

Определение 4. *Пусть дана временная система переходов* $\Im = (S, \Sigma, s_0, X, T)$. *Тогда*

- *Для непустых множеств* p,q $\in$ ***Conf***$(\Im)$ *будем писать* $p \xrightarrow{\sigma, \tau} q$,

  если $q = \{\langle s', \nu' \rangle \in Conf(\Im)\ \ |\ \exists\ \langle s, \nu \rangle\ \in\ p.\ \langle s, \nu \rangle \overset{\sigma,\tau}{\rightarrow} \langle s', \nu' \rangle\}$.
- *Определим множество* ***RS***$(\Im)$ *как наименьшее подмножество множества* $2^{Conf(\Im)} \setminus \{\varnothing\}$ *такое, что*
  - $\{C_0(\Im)\} \in \boldsymbol{RS}(\Im)$;
  - *Если* $p \in \boldsymbol{RS}(\Im)$ *и* $p \xrightarrow{\sigma, \tau} q$, *то* $q \in \boldsymbol{RS}(\Im)$.
- *Для любой конфигурации* $\langle s, \nu \rangle$ *определим множество*

  $A_\Im(\langle s, \nu \rangle) = \{(\sigma, \tau) \in \sum \times \mathrm{R}_+\ |\ \exists \langle s', \nu' \rangle \in Conf(\Im).\ \langle s, \nu \rangle \xrightarrow{\sigma,\tau} \langle s', \nu' \rangle\}$.
- *Для любого множества* p $\in$ ***RS***$(\Im)$ *зададим множество*

  $A_\Im(p) = \{A_\Im(\langle s, \nu \rangle)\ \ |\ \ \langle s, \nu \rangle \in p\}$.

Пример 3. Чтобы проиллюстрировать определенные выше понятия, рассмотрим временную систему переходов $\Im_1$, изображенную на рис. 1. Для этой системы переходов получаем, что

$$\boldsymbol{RS}(\Im_1) = \{\ \{\langle s_0, \nu_0 \rangle\},\ \{\langle s_1, \nu_1 \rangle\},\ \{\langle s_2, \nu_2 \rangle\}\ \ |\ \ \nu_0(x) = \nu_0(y) = 0,$$
$$\nu_1(x) \le 1,\ \nu_1(y) = 0,\ \nu_2(x) \le 3,\ \nu_2(y) \le 2\ \}.$$

Также, нетрудно заметить, что

$A_{\Im_1}(\{\langle s_0, \nu_0\rangle\}) = \{\{(a,t) \mid (t \le 1) \vee (4 < t \le 5) \vee (8 < t \le 9) \vee \ldots\}\}$,

$A_{\Im_1}(\{\langle s_1, \nu_1\rangle\}) = \{\{(b,t), (c,t') \mid (2 < t < 4) \vee$

$(6 < t < 9) \vee (10 < t < 13) \vee \ldots, (t' \le 3) \vee (4 < t' \le 7) \vee (8 < t' \le 11) \vee \ldots\}\}$ и

$A_{\Im_1}(\{\langle s_2, \nu_2\rangle\}) = \{\varnothing\}$.

3. КАТЕГОРИЯ ВРЕМЕННЫХ СИСТЕМ ПЕРЕХОДОВ $CTTS_{TEST}$

В этом разделе мы построим категорию временных систем переходов, состоящую из временных систем переходов и морфизмов между ними, а также приведем свойства этой категории.

Определение 5. *Пусть* $\Im = (S, \Sigma, s_0, X, T)$, $\Im' = (S', \Sigma', s'_0, X', T')$ *— две временные системы переходов. Отображение* μ *называется морфизмом между* $\Im$ *и* $\Im'$, *если* $\mu : RS(\Im) \to RS(\Im')$ *такое, что*

– $\mu(\{C_0(\Im)\}) = \{C_0(\Im')\}$;

– *если* $p_1 \xrightarrow{\sigma,\tau} p_2$ *в* $\Im$, *то* $\mu(p_1) \xrightarrow{\sigma,\tau} \mu(p_2)$ *в* $\Im'$;

– *для любого множества* $\alpha' \in A_{\Im'}(\mu(p))$ *существует множество* $\alpha \in A_{\Im}(p)$ *такое, что* $\alpha \subseteq \alpha'$.

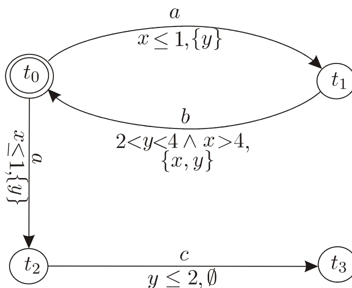


Рис. 2. Временная система переходов $\Im_2$

Пример 4. Для временной системы переходов $\Im_2$, изображенной на рис.2, выполнено следующее:

$\mathbf{RS}(\Im_2) = \{ \{\langle t_0, \nu'_0\rangle\}, \{\langle t_1, \nu'_1\rangle, \langle t_2, \nu'_1\rangle\}, \{\langle t_3, \nu'_2\rangle\} \mid \nu'_0(x) = \nu'_0(y) = 0,$

$\nu'_1(x) \le 1,\ \nu'_1(y) = 0,\ \nu'_2(x) \le 3,\ \nu'_2(y) \le 2 \}$.

А $A_{\Im_2}(\{\langle t_0, \nu'_0\rangle\}) = \{\{(a,t) \mid (t \le 1) \vee (4 < t \le 5) \vee (8 < t \le 9) \vee \ldots\}\}$,
$A_{\Im_2}(\{\langle t_1, \nu'_1\rangle, \langle t_2, \nu'_1\rangle\}) = \{\{(b,t) \mid (2 < t < 4) \vee$
$(6 < t < 9) \vee (10 < t < 13) \vee \ldots\}, \{(c,t) \mid (t \le 3) \vee$
$(4 < t \le 7) \vee (8 < t \le 11) \vee \ldots\}\}$ и $A_{\Im_2}(\{\langle t_3, \nu'_2\rangle\}) = \{\varnothing\}$.

Определим отображение μ_1 следующим образом:
$\mu_1(\{\langle t_0, \nu'_0\rangle\}) = \{\langle s_0, \nu_0\rangle\}$, $\mu_1(\{\langle t_1, \nu'_1\rangle, \langle t_2, \nu'_1\rangle\}) = \{\langle s_1, \nu_1\rangle\}$
и $\mu_1(\{\langle t_3, \nu'_2\rangle\}) = \{\langle s_2, \nu_2\rangle\}$.

Очевидно, что отображение μ_1 является морфизмом из временной системы переходов $\Im_2$ во временную систему переходов $\Im_1$, изображенную на рис. 1.

Теперь приведем формальное определение категории временных систем переходов $CTTS_{test}$.

Определение 6. *Категория $CTTS_{test}$ содержит временные системы переходов и морфизмы, определенные выше. При этом композиция двух морфизмов определена как обычная композиция функций, а тождественный морфизм — это тождественная функция.*

Далее докажем, что построенная выше категория временных систем переходов $CTTS_{test}$ обладает таким свойством, как коуниверсальность (pullbacks) [6, 8]. Приведем определение коуниверсальности.

Определение 7. *Категория M называется коуниверсальной, если для любой конструкции морфизмов $\Im_1 \xrightarrow{\mu_1} \Im_0 \xleftarrow{\mu_2} \Im_2$ существует конструкция $\Im_1 \xleftarrow{\pi_1} \Im \xrightarrow{\pi_2} \Im_2$ такая, что*

- $\mu_1 \circ \pi_1 = \mu_2 \circ \pi_2$;
- *для любой конструкции $\Im_1 \xleftarrow{\varphi_1} \Im' \xrightarrow{\varphi_2} \Im_2$ такой, что $\mu_1 \circ \varphi_1 = \mu_2 \circ \varphi_2$, существует морфизм $\zeta : \Im' \longrightarrow \Im$ такой, что $\varphi_1 = \pi_1 \circ \zeta$ и $\varphi_2 = \pi_2 \circ \zeta$.*

Теорема 1. *Категория $CTTS_{test}$ коуниверсальна в соответствии с определением 7.*

Доказательство. Пусть $\Im_1 \xrightarrow{\mu_1} \Im_0 \xleftarrow{\mu_2} \Im_2$ — конструкция морфизмов в категории $CTTS_{test}$, где $\Im_i = (S_i, \Sigma, s_0^i, X_i, T_i)$ для любого i = 0, 1 и 2.

Построим временную систему переходов $\mathfrak{I}_1\times\mathfrak{I}_2 = (S, \Sigma, s_0, X, T)$ следующим образом.

- $X = \{u\}$.
- $s_0 = (\{C_0(\mathfrak{I}_1)\}, \{C_0(\mathfrak{I}_2)\}, A_{\mathfrak{I}_1}(\{C_0(\mathfrak{I}_1)\})\cap A_{\mathfrak{I}_2}(\{C_0(\mathfrak{I}_2)\}))$.
- S — наименьшее подмножество множества $\boldsymbol{RS}(\mathfrak{I}_1)\times\boldsymbol{RS}(\mathfrak{I}_1)\times 2^{(\Sigma\times\mathbf{R}_+)}$, удовлетворяющее следующим условиям:
  - $s_0 \in S$;
  - Пусть $(p,q,D)\in S$ и пусть $(\sigma,\tau)\in D$, $p\xrightarrow{\sigma,\tau}p'$ и $q\xrightarrow{\sigma,\tau}q'$. Тогда верно, что $(p',q',D')\in S$ для всех $D'\in M(p',q')$, где
    $M(p',q') = \{\alpha\cap(\bigcup_{\beta\in A_{\mathfrak{I}_2}(q')}\beta) \mid \alpha\in A_{\mathfrak{I}_1}(p')\} \cup \{\beta\cap(\bigcup_{\alpha\in A_{\mathfrak{I}_1}(p')}\alpha) \mid \beta\in A_{\mathfrak{I}_2}(q')\}$.
- $((p,q,D), \sigma, \{u=\tau\}, \varnothing, (p',q',D')) \in T \Leftrightarrow (\sigma,\tau)\in D$, $p\xrightarrow{\sigma,\tau}p'$, $q\xrightarrow{\sigma,\tau}q'$ и $D'\in M(p',q')$.

По построению очевидно, что $\mathfrak{I}_1\times\mathfrak{I}_2$ является временной системой переходов. Нетрудно проверить, что для $\mathfrak{I}_1\times\mathfrak{I}_2$ выполнены следующие полезные свойства:

- для $(p,q,D)\in S$ и временной функции прогресса ν множество $A_{\mathfrak{I}_1\times\mathfrak{I}_2}(\langle(p,q,D),\nu\rangle) = D$;
- $L(\mathfrak{I}_1\times\mathfrak{I}_2) = L(\mathfrak{I}_1)\cap L(\mathfrak{I}_2)$;
- для любого $Z\in \boldsymbol{RS}(\mathfrak{I}_1\times\mathfrak{I}_2)$ существуют множества $p\in \boldsymbol{RS}(\mathfrak{I}_1)$ и $q\in \boldsymbol{RS}(\mathfrak{I}_2)$ и функция ν такие, что $Z = \{\langle(p,q,D),\nu\rangle \mid D\in M(p,q)\}$ и $\nu(u) = \tau$.

Теперь определим отображения $\pi_i : \mathfrak{I}_1\times\mathfrak{I}_2 \longrightarrow \mathfrak{I}_i$, $(i=1,2)$ следующим образом. Пусть $Z\in \boldsymbol{RS}(\mathfrak{I}_1\times\mathfrak{I}_2)$. Тогда верно $\pi_1(Z) = p$ и $\pi_2(Z) = q$, где p и q определяются множеством Z, поскольку по сказанному выше имеем, что $Z = \{\langle(p,q,D),\nu\rangle \mid D\in M(p,q)\}$. Проверим, что π_1 является морфизмом (проверка этого же факта для π_2 аналогична). По определению очевидно, что $\pi_1 : \mathrm{RS}(\mathfrak{I}_1\times\mathfrak{I}_2)\longrightarrow \mathrm{RS}(\mathfrak{I}_1)$. Проверим выполнение трех условий из определения 5.

- По построению имеем, что $\pi_1(\{C_0(\mathfrak{I}_1\times\mathfrak{I}_2)\}) = \{C_0(\mathfrak{I}_1)\}$.
- Теперь пусть $Z_1\xrightarrow{\sigma,\tau}Z_2$ *в* $\mathfrak{I}_1\times\mathfrak{I}_2$. Тогда по доказанному выше имеем, что

$Z_1 = \{\langle(p_1, q_1, D), \nu_1\rangle \mid D \in M(p_1,q_1)\}$ и
$Z_2 = \{\langle(p_2, q_2, D), \nu_2\rangle \mid D \in M(p_2,q_2)\}$.

По определению отношения $\xrightarrow{\sigma,\tau}$ получаем, что $\langle(p_1, q_1, D_1), \nu_1\rangle \xrightarrow{\sigma,\tau} \langle(p_2, q_2, D_2), \nu_2\rangle$ для некоторых $\langle(p_1, q_1, D_1), \nu_1\rangle \in Z_1$ и $\langle(p_2, q_2, D_2), \nu_2\rangle \in Z_2$. По построению $\Im_1 \times \Im_2$ это означает, что $p_1 \xrightarrow{\sigma,\tau} p_2$. Таким образом, $\pi_1(Z_1) \xrightarrow{\sigma,\tau} \pi_1(Z_2)$ в $\Im_1$.

- Пусть теперь $\alpha_1 \in A_{\Im_1}(\pi_1(Z))$. Вновь воспользуемся тем, что $Z = \{\langle(p,q,D),\nu\rangle \mid D \in M(p,q)\}$. По определению множества $M(p,q)$ получаем, что $\langle(p,q, (\alpha_1 \cap (\bigcup_{\beta \in A_{\Im_2}(q)} \beta)), \nu\rangle \in Z$. По доказанному выше заключаем, что $A_{\Im_1 \times \Im_2}(\langle(p,q, (\alpha_1 \cap (\bigcup_{\beta \in A_{\Im_2}(q)} \beta)), \nu\rangle) = \alpha_1 \cap (\bigcup_{\beta \in A_{\Im_2}(q)} \beta)$.
  Это означает, что $\alpha_1 \cap (\bigcup_{\beta \in A_{\Im_2}(q)} \beta) \in A_{\Im_1 \times \Im_2}(Z)$ и $\alpha_1 \cap (\bigcup_{\beta \in A_{\Im_2}(q)} \beta) \subseteq \alpha_1$.

Таким образом, мы показали, что π_1 является морфизмом. Из определения морфизма и множества ***RS***$(\Im_1 \times \Im_2)$ нетрудно заметить, что $\mu_1 \circ \pi_1 = \mu_2 \circ \pi_2$.

Теперь, пусть $\Im_1 \xleftarrow{\varphi_1} \Im' \xrightarrow{\varphi_2} \Im_2$ — конструкция морфизмов, удовлетворяющая условию: $\mu_1 \circ \varphi_1 = \mu_2 \circ \varphi_2$. Кроме того, пусть $V \in$ ***RS***$(\Im')$ такое, что $\{C_0(\Im')\} \xrightarrow{\sigma_1,\tau_1} \ldots \xrightarrow{\sigma_n,\tau_n} V$. Тогда определим отображение $\zeta : \mathrm{RS}(\Im') \longrightarrow \mathrm{RS}(\Im_1 \times \Im_2)$ по правилу: $\zeta(V) = \{\langle(\varphi_1(V), \varphi_2(V), D), \nu\rangle \mid D \in M(\varphi_1(V), \varphi_2(V)), \nu(u) = \tau_n\}$. Докажем, что определенное отображение является морфизмом. Для этого достаточно проверить выполнение трех условий из определения 5.

- Пусть $\{C_0(\Im')\} = V_0$. Нетрудно заметить, что $\zeta(V_0) = \{\langle(\varphi_1(V_0), \varphi_2(V_0), D), \nu\rangle \mid D \in M(\varphi_1(V_0), \varphi_2(V_0)), \nu(u) = 0\} = \{C_0(\Im_1 \times \Im_2)\}$.
- Теперь пусть $V_1 \xrightarrow{\sigma,\tau} V_2$ *в* $\Im'$. Так как φ_1 и φ_2 — морфизмы, получаем, что $\varphi_1(V_1) \xrightarrow{\sigma,\tau} \varphi_1(V_2)$ и $\varphi_2(V_1) \xrightarrow{\sigma,\tau} \varphi_2(V_2)$. Тогда по доказанным выше фактам и по построению $\Im_1 \times \Im_2$ имеем, что $\varsigma(V_1) \xrightarrow{\sigma,\tau} \varsigma(V_2)$ *в* $\Im_1 \times \Im_2$.
- Пусть теперь $\chi \in A_{\Im_1 \times \Im_2}(\varsigma(V))$. По доказанному выше имеем, что $A_{\Im_1 \times \Im_2}(\varsigma(V)) = \{D \mid D \in M(\varphi_1(V), \varphi_2(V))\}$. Отсюда получаем, что

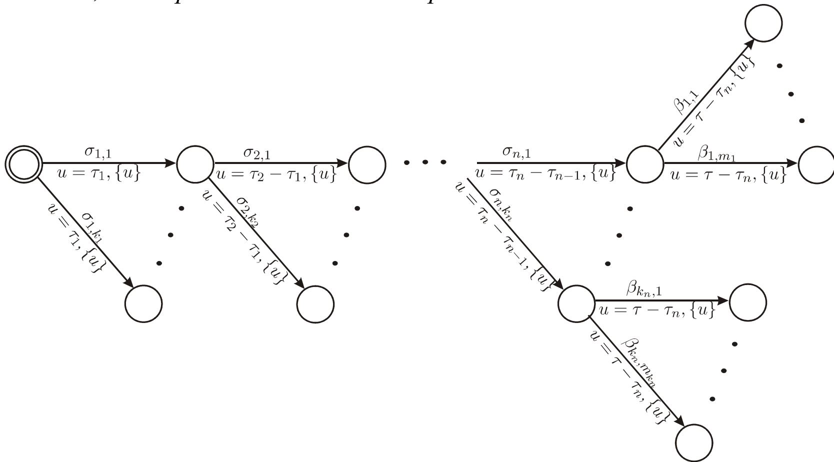
$\chi \in M(\varphi_1(V), \varphi_2(V))$. Без ограничения общности можно считать, что $\chi = \alpha \bigcap (\bigcup_{\beta \in A_{\Im_2}(\varphi_2(V))} \beta))$ для некоторого $\alpha \in A_{\Im_1}(\varphi_1(V))$. Так как φ_1 — морфизм, то существует множество $\alpha' \in A_{\Im'}(V)$ такое, что $\alpha' \subseteq \alpha$. Далее, так как φ_2 является морфизмом, получаем, что $\alpha' \in \bigcup_{\beta \in A_{\Im_2}(\varphi_2(V))} \beta$. Тогда ясно, что $\alpha' \subseteq \chi$.

Таким образом, мы показали, что ζ является морфизмом. Равенства $\varphi_1 = \pi_1 \circ \zeta$ и $\varphi_2 = \pi_2 \circ \zeta$ следуют из определения морфизмов ζ, π_1, π_2. □

4. ОТКРЫТЫЙ МОРФИЗМ

В данном разделе определяется подкатегория наблюдений P_{test} для категории временных систем переходов $CTTS_{test}$, по подкатегории P_{test} строится открытый морфизм, а в завершении доказывается критерий открытости для морфизмов из категории $CTTS_{test}$.

Определение 8. *Подкатегория P_{test} в категории $CTTS_{test}$ содержит наблюдения, т. е. временные системы переходов вида:*



и морфизмы между ними.

Далее для категории $CTTS_{test}$ по выделенной подкатегории P_{test} можно определить понятие открытого морфизма, следуя общей схеме, предложенной в работе [7].

Определение 9. *Морфизм* $\mu : \mathfrak{I} \to \mathfrak{I}'$ *в категории* $CTTS_{test}$ *называется открытым тогда и только тогда, когда из существования морфизмов* $\mu_1 : \mathcal{O}' \to \mathfrak{I}'$, $\mu_2 : \mathcal{O} \to \mathfrak{I}$ *в категории* $CTTS_{test}$ *и морфизма* $\mu_3 : \mathcal{O} \to \mathcal{O}'$ *в подкатегории* P_{test} *таких, что* $\mu \circ \mu_2 = \mu_1 \circ \mu_3$, *следует существование морфизма* $\mu_4 : \mathcal{O}' \to \mathfrak{I}$ *в категории* $CTTS_{test}$ *такого, что* $\mu_1 = \mu \circ \mu_4$ *и* $\mu_2 = \mu_4 \circ \mu_3$.

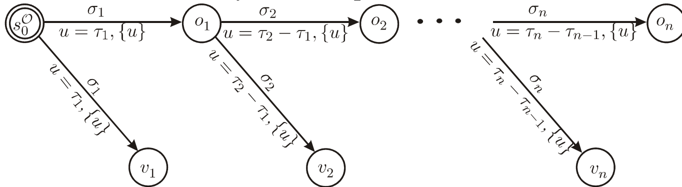
Теперь приведем критерий открытости для морфизмов в категории $CTTS_{test}$..

Теорема 2. *Пусть* $\mathfrak{I}$ *и* $\mathfrak{I}'$ *— временные системы переходов и* μ *— морфизм из* $\mathfrak{I}$ *в* $\mathfrak{I}'$. *Морфизм* μ *открыт тогда и только тогда, когда выполнены следующие два условия:*

- *если для некоторых* $p_1 \in \boldsymbol{RS}(\mathfrak{I})$, $q_2 \in \boldsymbol{RS}(\mathfrak{I}_2)$, $\beta \in \Sigma$ *и* $\tau \in \boldsymbol{R}_+$ *верно, что* $\mu(p_1) \xrightarrow{\beta,\tau} q_2$, *то существует* $p_2 \in \boldsymbol{RS}(\mathfrak{I})$ *такое, что* $p_1 \xrightarrow{\beta,\tau} p_2$ *и* $\mu(p_2) = q_2$;
- *для любого множества* $\alpha \in A_{\mathfrak{I}}(p_1)$ *существует множество* $\alpha' \in A_{\mathfrak{I}'}(\mu(p_1))$ *такое, что* $\alpha' \subseteq \alpha$.

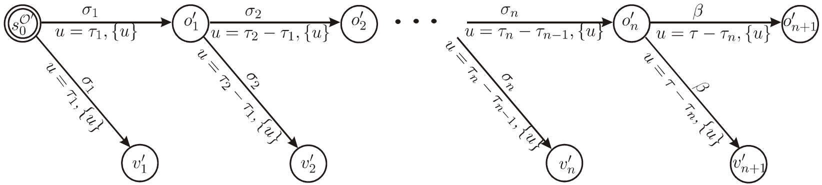
Доказательство. ($\Rightarrow$) Пусть μ — открытый морфизм. Для начала проверим выполнение первого условия теоремы.

Пусть $p_1 \in \boldsymbol{RS}(\mathfrak{I})$, $q_2 \in \boldsymbol{RS}(\mathfrak{I}_2)$, $\beta \in \Sigma$, $\tau \in \boldsymbol{R}_+$ и $\mu(p_1) \xrightarrow{\beta,\tau} q_2$. По построению множества $\boldsymbol{RS}(\mathfrak{I})$ получаем существование последовательности $p^0 \xrightarrow{\sigma_1,\tau_1} p^1 \xrightarrow{\sigma_2,\tau_2} \ldots \xrightarrow{\sigma_n,\tau_n} p^n$ такой, что $p^0 = \{C_0(\mathfrak{I})\}$ и $p^n = p_1$. Для $i = 1..n$ обозначим через q^i множества $\mu(p^i)$ и, кроме того, пусть $q^{n+1} = q_2$. Определим наблюдение $\mathcal{O}$ следующим образом.



Легко видеть, что $\mathbf{RS}(\mathcal{O})=\{Z_0,Z_1,\ldots,Z_n \mid Z_0=\{C_0(\mathcal{O})\}, Z_i=\{\langle o_i,\nu_i\rangle,\langle v_i,\nu_i\rangle\}$ для любого $i = 1..n$, $\nu_1(u) = \tau_1$, $\nu_j(u) = \tau_j - \tau_{j-1}$ для всех $j = 2..n\}$. Кроме то-

го, очевидно, что $A_{\mathscr{O}}(Z_{i-1}) = \{\{(\sigma_i,\tau_i)\},\ \varnothing\}$ для i = 1..n и $A_{\mathscr{O}}(Z_n) = \{\varnothing\}$. Теперь определим еще одно наблюдение $\mathscr{O}'$:



Для этого наблюдения верно, что $\boldsymbol{RS}(\mathscr{O}')=\{ Z'_0, Z'_1,\ldots, Z'_n, Z'_{n+1}$ | $Z'_0=\{C_0(\mathscr{O}')\}$, $Z'_i=\{ \langle o'_i, \nu'_i \rangle, \langle v'_i, \nu'_i \rangle \}$ для любого i =1..(n+1), $\nu'_1(u) = \tau_1$, $\nu'_j(u) = \tau_j - \tau_{j-1}$ для всех j = 2..n и $\nu'_{n+1}(u) = t - t_n$ }. Кроме того, имеем:

$A_{\mathscr{O}'}(Z'_{i-1}) = \{\{(\sigma_i,\tau_i)\},\ \varnothing\}$ для i = 1..n,

$A_{\mathscr{O}'}(Z'_n) = \{\{(\beta,\tau)\},\ \varnothing\}$ и

$A_{\mathscr{O}'}(Z'_{n+1}) = \{\varnothing\}$.

Определим три отображения $\mu_1 : \mathscr{O} \to \mathscr{O}'$, $\mu_2 : \mathscr{O} \to \mathfrak{I}$ и $\mu_3 : \mathscr{O}' \to \mathfrak{I}'$ по следующим правилам: $\mu_1(Z_i) = Z'_i$, $\mu_2(Z_i) = p^i$ и $\mu_3(Z'_j) = q^j$ для любого $i = 0..n$ и $j = 0..(n+1)$. Легко видеть, что три вышеопределенных отображения являются морфизмами в соответствии с определением 5. Кроме того, очевидно, что $\mu \circ \mu_2 = \mu_3 \circ \mu_1$.

Так как μ является открытым морфизмом, то по определению 9 существует морфизм $\mu' : \mathscr{O}' \to \mathfrak{I}$ такой, что $\mu_2 = \mu' \circ \mu_1$ и $\mu_3 = \mu \circ \mu'$. Заметим, что $\mu'(Z'_n) = \mu' \circ \mu_1 (Z_n) = \mu_2(Z_n) = p_1$ и $\mu \circ \mu'(Z'_{n+1}) = \mu_3(Z'_{n+1}) = q_2$. Далее, так как $Z'_n \xrightarrow{\beta,\tau} Z'_{n+1}$ и μ' является морфизмом, то $\mu'(Z'_{n+1}) \in \boldsymbol{RS}(\mathfrak{I}')$ и $\mu'(Z'_n) \xrightarrow{\beta,\tau} \mu'(Z'_{n+1})$. Пусть $p_2 = \mu'(Z'_{n+1})$. Тогда очевидно, что $p_1 \xrightarrow{\beta,\tau} p_2$ и $\mu(p_2) = q_2$, что и требовалось показать.

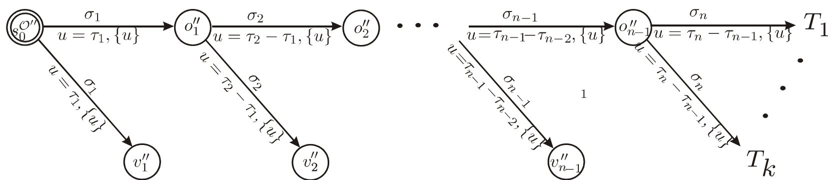
Осталось проверить выполнение второго условия теоремы.

Пусть $\alpha \in A_{\mathfrak{I}}(p_1)$. Как и ранее по построению множества $\boldsymbol{RS}(\mathfrak{I})$ получаем существование последовательности

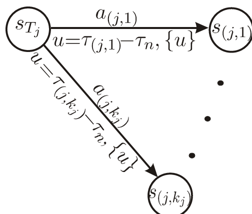
$$p^0 \xrightarrow{\sigma_1,\tau_1} p^1 \xrightarrow{\sigma_2,\tau_2} \ldots \xrightarrow{\sigma_n,\tau_n} p^n$$

такой, что $p^0 = \{C_0(\mathfrak{I})\}$ и $p^n = p_1$. Вновь для i =1..n обозначим через q^i множества $\mu(p^i)$ и, кроме того, пусть $q^{n+1} = q_2$. Определим наблюдение $\mathscr{O}$ так же, как и ранее. Пусть $A_{\mathfrak{I}'}(q^n) = \{A_1, \ldots A_k\}$, где

$A_j = \{(a_{(j,1)}, \tau_{(j,1)}), \ldots, (a_{(j,k_j)}, \tau_{(j,k_j)})\}$ для любого j =*1..k*. Теперь определим еще одно наблюдение $\mathcal{O}$":



Здесь подсистемы T_j (*j=1..k*) имеют следующий вид:



Пусть $\bigcup_{j=1}^{k} A_j = \{(a^1, \tau^1), \ldots, (a^l, \tau^l)\}$. Определим Z_i^* как максимальные подмножества множества $\{\langle s_{(i,j)}, \nu"_{(i,j)}\rangle \mid \nu"_{(i,j)} = \tau_{(i,j)} - \tau_n,$ i = *1..k*, j = *1..k$_i$*$\}$ такие, что $\{\langle s_{T_1}, \nu"_n\rangle, \ldots, \langle s_{T_k}, \nu"_n\rangle\} \xrightarrow{a^i, \tau^i} Z_i^*$ i = *1..l*. С учетом этих определений из построения наблюдения $\mathcal{O}$" получаем, что

RS($\mathcal{O}$")=$\{ Z"_0, Z"_1, \ldots, Z"_n, Z_1^*, \ldots Z_l^* \mid Z''_0 = \{C_0(\mathcal{O}')\}, Z''_i =$
$\{ \langle o"_i, \nu"_i \rangle, \langle v''_i, \nu"_i \rangle \}$ для любого i = *1..(n-1)*, $Z''_n =$
$\{ \langle s_{T_1}, \nu"_n \rangle, \ldots, \langle s_{T_k}, \nu"_n \rangle \}$, $\nu"_1(u) = \tau_1$, $\nu"_j(u) = \tau_j - \tau_{j-1}$ для всех j = *2..n*$\}$.

Кроме того, имеем
$A_{\mathcal{O}"}(Z"_{i-1}) = \{\{(\sigma_i, \tau_i)\}, \varnothing\}$ для i = *1..n,*
$A_{\mathcal{O}"}(Z"_n) = A_{\mathfrak{I}'}(q^n)$ и
$A_{\mathcal{O}"}(Z_i^*) = \{\varnothing\}$ для всех i = *1..l*.
Поскольку $A_{\mathcal{O}"}(Z"_n) = A_{\mathfrak{I}'}(q^n)$, то для любой пары (a^i, τ^i) существует $q_i^* \in$ ***RS***($\mathfrak{I}'$) такое, что $q_n \xrightarrow{a^i, \tau^i} q_i^*$.

Определим три отображения $\mu'_1 : \mathcal{O} \to \mathcal{O}''$, $\mu'_2 : \mathcal{O} \to \mathfrak{I}$ и $\mu'_3 : \mathcal{O}'' \to \mathfrak{I}'$ по следующим правилам: $\mu'_1(Z_j) = Z''_j$, $\mu'_2(Z_j) = p^j$, $\mu'_3(Z''_j) = q^j$ и $\mu'_3(Z_i^*) = q_i^*$ для любого $i = 1..l$ и $j = 0..n$. Нетрудно заметить, что эти отображения являются морфизмами в соответствии с определением 5. Кроме того, очевидно, что $\mu \circ \mu'_2 = \mu'_3 \circ \mu'_1$.

Так как μ является открытым морфизмом, то по определению 9 существует морфизм $\mu'' : \mathcal{O}'' \to \mathfrak{I}$ такой, что $\mu'_2 = \mu'' \circ \mu'_1$ и $\mu'_3 = \mu \circ \mu''$. Заметим, что $\mu''(Z''_n) = \mu'' \circ \mu'_1 (Z_n) = \mu'_2(Z_n) = p_1$. По условию теоремы $\alpha \in A_{\mathfrak{I}}(p_1)$. Далее, так как μ'' является морфизмом, то существует $\alpha' \in A_{\mathcal{O}''}(Z''_n)$ такое, что $\alpha' \subseteq \alpha$. Однако $A_{\mathcal{O}''}(Z''_n) = A_{\mathfrak{I}'}(q^n) = A_{\mathfrak{I}'}(\mu(p_1))$, что завершает доказательство.

($\Leftarrow$) Пусть выполнены оба условия теоремы. Докажем, что морфизм μ открыт. Для этого воспользуемся определением 9. Пусть $\mu_1 : \mathcal{O}' \to \mathfrak{I}'$, $\mu_2 : \mathcal{O} \to \mathfrak{I}$ — морфизмы в категории $CTTS_{test}$ и $\mu_3 : \mathcal{O} \to \mathcal{O}'$ — морфизм в подкатегории P_{test} такие, что $\mu \circ \mu_2 = \mu_1 \circ \mu_3$. Определим отображение $\mu' : \mathcal{O}' \to \mathfrak{I}$ по индукции следующим образом.

- $\mu'(\{C_0(\mathcal{O}')\}) = \{C_0(\mathfrak{I})\}$. Кроме того, ясно, что $\mu \circ \mu'(\{C_0(\mathcal{O}')\}) = \mu_1(\{C_0(\mathcal{O}')\})$.
- Пусть $\mu'(Z)$ уже определено и при этом $Z \xrightarrow{\sigma,\tau} Z'$, а $\mu'(Z')$ еще не определено. По предположению индукции $\mu \circ \mu'(Z) = \mu_1(Z)$. Тогда, так как μ_1 – морфизм, то $\mu_1(Z) \xrightarrow{\sigma,\tau} \mu_1(Z')$. Отсюда по первому условию теоремы получаем существование $p \in \boldsymbol{RS}(\mathfrak{I})$ такого, что $\mu'(Z) \xrightarrow{\sigma,\tau} p$ и $\mu(p) = \mu_1(Z')$. По определению множества $\boldsymbol{RS}(\mathfrak{I})$ получаем, что такое p единственно. Тогда положим, что $\mu'(Z') = p$. Заметим, что по второму условию теоремы для любого множества $\alpha \in A_{\mathfrak{I}}(p)$ существует множество $\alpha' \in A_{\mathfrak{I}'}(\mu(p)) = A_{\mathfrak{I}'}(\mu_1(Z'))$ такое, что $\alpha' \subseteq \alpha$. Но μ_1 является морфизмом, поэтому для любого $\alpha' \in A_{\mathfrak{I}'}(\mu_1(Z'))$ существует множество $\alpha'' \in A_{\mathcal{O}'}(Z')$ такое, что $\alpha'' \subseteq \alpha'$.

Таким образом, очевидно, что μ' является морфизмом. Кроме того, нетрудно заметить, что $\mu_2 = \mu' \circ \mu_1$ и $\mu_3 = \mu \circ \mu'$. По определению 9 заключаем, что μ — открытый морфизм. □

5. ХАРАКТЕРИЗАЦИЯ ТЕСТОВОЙ ЭКВИВАЛЕНТНОСТИ

В данном разделе по выделенной подкатегории определяется абстрактная эквивалентность, приводится определение временной тестовой эквивалентности и доказывается совпадение этих эквивалентностей.

Определение 10. *Две временные системы переходов* $\Im$ *и* $\Im'$ *называются* P_{test}*-эквивалентными, если и только если существует конструкция открытых морфизмов* $\Im \xleftarrow{\mu} \Im^* \xrightarrow{\mu'} \Im'$ *с вершиной* $\Im^*$.

Корректность этого определения следует из коуниверсальности категории $CTTS_{test}$. Далее приведем понятие временной тестовой эквивалентности.

Определение 11. *Две временные системы переходов* $\Im_1$ *и* $\Im_2$ *называются тестово эквивалентными, если и только если для любого временного слова* $(\sigma_1,\tau_1)\ldots(\sigma_n,\tau_n)$ *и любого подмножества* $L \subseteq \Sigma \times R_+$ *выполнено:* $\Im_1$ **after** $(\sigma_1,\tau_1)\ldots(\sigma_n,\tau_n)$ **MUST** $L \Leftrightarrow \Im_2$ **after** $(\sigma_1,\tau_1)\ldots(\sigma_n,\tau_n)$ **MUST** L. *Здесь запись* $\Im_i$ **after** $(\sigma_1,\tau_1)\ldots(\sigma_n,\tau_n)$ **MUST** L $(i=1,2)$ *означает, что для любого* $\langle s,\nu\rangle \in \boldsymbol{Conf}(\Im_i)$ *такого, что* $C_0(\Im_i)\xrightarrow{\sigma_1,\tau_1}\ldots\xrightarrow{\sigma_n,\tau_n}\langle s,\nu\rangle$, *существуют* $(a,\tau) \in L$ *и* $\langle s',\nu'\rangle \in \boldsymbol{Conf}(\Im_i)$ *такие, что* $\langle s,\nu\rangle \xrightarrow{a,\tau} \langle s',\nu'\rangle$.

Пример 5. Для временных систем переходов, изображенных на рис. 3, верно, что $\Im_3$ и $\Im_4$ являются тестово эквивалентными, а $\Im_4$ и $\Im_5$ — нет, так как $\Im_5$ **after** *(a,1)* **MUST** $\{(c,2)\}$, что неверно для $\Im_4$.

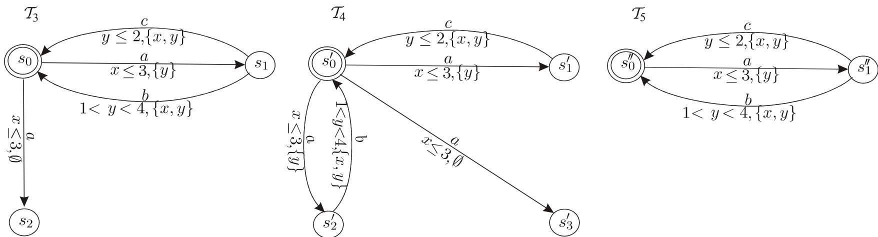


Рис. 3. Временные системы переходов $\Im_3$, $\Im_4$ и $\Im_5$

Теперь можно сформулировать основной результат данной статьи.

Теорема 3. *Временные системы переходов $\mathfrak{I}$ и $\mathfrak{I}'$ P_{test}-эквивалентны, если и только если они тестово эквивалентны.*

Доказательство. ($\Rightarrow$) Пусть $\mathfrak{I}$ и $\mathfrak{I}'$ P_{test}-эквивалентны, тогда по определению существует конструкция открытых морфизмов $\mathfrak{I} \xleftarrow{\mu} \mathfrak{I}^* \xrightarrow{\mu'} \mathfrak{I}'$. В силу транзитивности эквивалентности достаточно показать, что $\mathfrak{I}$ и $\mathfrak{I}^*$ тестово эквивалентны. Заметим, что $L(\mathfrak{I}) = L(\mathfrak{I}^*)$ в силу открытости морфизма μ. Пусть $(\sigma_1,\tau_1)\ldots(\sigma_n,\tau_n)$ — некоторое временное слово, $L \subseteq \Sigma \times R_+$ и $\mathfrak{I}$ **after** $(\sigma_1,\tau_1)\ldots(\sigma_n,\tau_n)$ **MUST** L. Если $(\sigma_1,\tau_1)\ldots(\sigma_n,\tau_n) \notin L(\mathfrak{I}^*)$, то очевидно, что $\mathfrak{I}^*$ **after** $(\sigma_1,\tau_1)\ldots(\sigma_n,\tau_n)$ **MUST** L. Пусть $(\sigma_1,\tau_1)\ldots(\sigma_n,\tau_n) \in L(\mathfrak{I}^*) = L(\mathfrak{I})$. Это гарантирует существование $p \in \boldsymbol{RS}(\mathfrak{I})$ такого, что $\{C_0(\mathfrak{I})\} \xrightarrow{\sigma_1,\tau_1} \ldots \xrightarrow{\sigma_n,\tau_n} p$. Далее, так как $\mathfrak{I}$ **after** $(\sigma_1,\tau_1)\ldots(\sigma_n,\tau_n)$ **MUST** L, то для любого множества $\alpha \in A_{\mathfrak{I}}(p)$ верно, что $\alpha \cap L \neq \varnothing$. Теперь воспользуемся тем, что μ является открытым морфизмом. По теореме 2 получаем существование $q \in \boldsymbol{RS}(\mathfrak{I}^*)$ такого, что $\mu(q)=p$, $\{C_0(\mathfrak{I}^*)\} \xrightarrow{\sigma_1,\tau_1} \ldots \xrightarrow{\sigma_n,\tau_n} q$ и для любого множества $\alpha' \in A_{\mathfrak{I}^*}(q)$ существует множество $\alpha \in A_{\mathfrak{I}}(p)$ такое, что $\alpha \subseteq \alpha'$. Отсюда получаем, что для любого множества $\alpha' \in A_{\mathfrak{I}^*}(q)$ верно, что $\alpha' \cap L \neq \varnothing$. Таким образом, мы показали, что $\mathfrak{I}^*$ **after** $(\sigma_1,\tau_1)\ldots(\sigma_n,\tau_n)$ **MUST** L.

Теперь пусть $\mathfrak{I}^*$ **after** $(\sigma_1,\tau_1)\ldots(\sigma_n,\tau_n)$ **MUST** L для некоторого временного слова $(\sigma_1,\tau_1)\ldots(\sigma_n,\tau_n)$ и некоторого множества $L \subseteq \Sigma \times \boldsymbol{R}_+$. Если $(\sigma_1,\tau_1)\ldots(\sigma_n,\tau_n) \notin L(\mathfrak{I})$, то по определению предиката ясно, что $\mathfrak{I}$ **after** $(\sigma_1,\tau_1)\ldots(\sigma_n,\tau_n)$ **MUST** L. Пусть $(\sigma_1,\tau_1)\ldots(\sigma_n,\tau_n) \in L(\mathfrak{I})=L(\mathfrak{I}^*)$. Это означает, что существует $q \in \boldsymbol{RS}(\mathfrak{I}^*)$ такое, что $\{C_0(\mathfrak{I}^*)\} \xrightarrow{\sigma_1,\tau_1} \ldots \xrightarrow{\sigma_n,\tau_n} q$ и для любого множества $\alpha' \in A_{\mathfrak{I}^*}(q)$ верно, что $\alpha' \cap L \neq \varnothing$. Так как μ является морфизмом, то определению 5 имеем, что

$$\mu(q) \in \boldsymbol{RS}(\mathfrak{I}),\ \{C_0(\mathfrak{I})\} \xrightarrow{\sigma_1,\tau_1} \ldots \xrightarrow{\sigma_n,\tau_n} \mu(q)$$

и для любого множества $\alpha \in A_{\mathfrak{I}}(\mu(q))$ существует множество $\alpha' \in A_{\mathfrak{I}^*}(q)$ такое, что $\alpha' \subseteq \alpha$. Суммируя вышесказанное, получаем, что для любого множества $\alpha \in A_{\mathfrak{I}}(\mu(q))$ верно, что $\alpha \cap L \neq \varnothing$. Это означает, что $\mathfrak{I}$ **after** $(\sigma_1,\tau_1)\ldots(\sigma_n,\tau_n)$ **MUST** L.

Таким образом, временные системы переходов $\mathfrak{I}$ и $\mathfrak{I}^*$ являются тестово эквивалентными, что и требовалось доказать.

(⇐) Пусть теперь $\mathfrak{I}$ и $\mathfrak{I}'$ тестово эквивалентны. Нетрудно проверить, что $L(\mathfrak{I}) = L(\mathfrak{I}')$. Построим временную систему переходов $\mathfrak{I}\times\mathfrak{I}'$ и морфизмы π_1 и π_2 так, как это было сделано в доказательстве теоремы 1. Для завершения доказательства необходимо проверить, что морфизмы π_1 и π_2 открыты. Проверим открытость морфизма π_1 по теореме 2 (доказательство открытости π_2 аналогично). Для этого необходимо доказать выполнение двух условий теоремы 2.

- Пусть $Z\in \boldsymbol{RS}(\mathfrak{I}\times\mathfrak{I}')$, $p'\in \boldsymbol{RS}(\mathfrak{I})$, $\beta\in\Sigma$, $\tau\in \boldsymbol{R}_+$ и $\pi_1(Z)\xrightarrow{\beta,\tau} p'$. Без ограничения общности считаем, что

$$\{C_0(\mathfrak{I})\} \xrightarrow{\sigma_1,\tau_1} \ldots \xrightarrow{\sigma_n,\tau_n} \pi_1(Z) \xrightarrow{\beta,\tau} p'.$$

  Это означает, что $(\sigma_1,\tau_1)\ldots(\sigma_n,\tau_n)(\beta,\tau)\in L(\mathfrak{I})$. В доказательстве теоремы 1 было показано, что $L(\mathfrak{I}\times\mathfrak{I}') = L(\mathfrak{I})\bigcap L(\mathfrak{I}')$, но в нашем случае $L(\mathfrak{I}) = L(\mathfrak{I}')$, следовательно, $(\sigma_1,\tau_1)\ldots(\sigma_n,\tau_n)(\beta,\tau)\in L(\mathfrak{I}\times\mathfrak{I}')$. Отсюда, учитывая определение множества $\boldsymbol{RS}(\mathfrak{I}\times\mathfrak{I}')$, имеем, что $\{C_0(\mathfrak{I}\times\mathfrak{I}')\} \xrightarrow{\sigma_1,\tau_1} \ldots \xrightarrow{\sigma_n,\tau_n} Z \xrightarrow{\beta,\tau} Z'$. Кроме того, нетрудно видеть, что $\pi_1(Z') = p'$.
- Пусть $\chi\in A_{\mathfrak{I}\times\mathfrak{I}'}(Z)$. По построению $\mathfrak{I}\times\mathfrak{I}'$ имеем, что $\chi\in M(\pi_1(Z),\pi_2(Z))$. Без ограничения общности считаем, что $\chi=\overline{\alpha}\bigcap(\bigcup_{\beta\in A_{\mathfrak{I}'}(\pi_2(Z))}\beta)$. Поскольку $\mathfrak{I}$ и $\mathfrak{I}'$ тестово эквиваленты, то для любого множества $\alpha\in A_{\mathfrak{I}}(\pi_1(Z))$ существует множество $\beta_\alpha\in A_{\mathfrak{I}'}(\pi_2(Z))$ такое, что $\beta_\alpha\subseteq\alpha$, и наоборот, для любого множества $\beta\in A_{\mathfrak{I}'}(\pi_2(Z))$ существует множество $\alpha_\beta\in A_{\mathfrak{I}}(\pi_1(Z))$ такое, что $\alpha_\beta\subseteq\beta$. Тогда для $\overline{\alpha}$ существует множество $\beta_{\overline{\alpha}}\in A_{\mathfrak{I}'}(\pi_2(Z))$ такое, что $\beta_{\overline{\alpha}}\subseteq\overline{\alpha}$. Далее получаем существование множества $\alpha_{\beta_{\overline{\alpha}}}\in A_{\mathfrak{I}}(\pi_1(Z))$ такого, что $\alpha_{\beta_{\overline{\alpha}}}\subseteq\beta_{\overline{\alpha}}$. Очевидно, что

$$\alpha_{\beta_{\overline{\alpha}}}\subseteq\beta_{\overline{\alpha}}\subseteq(\bigcup_{\beta\in A_{\mathfrak{I}'}(\pi_2(Z))}\beta) \text{ и } \alpha_{\beta_{\overline{\alpha}}}\subseteq\beta_{\overline{\alpha}}\subseteq\overline{\alpha}.$$

  Значит, верно, что $\alpha_{\beta_{\overline{\alpha}}}\subseteq\chi$.

Таким образом, по теореме 2 получаем, что морфизм π_1 открыт. □

6. ЗАКЛЮЧЕНИЕ

В этой работе была показана применимость теории открытых морфизмов [7] для исследований временных вариантов тестовых эквивалентностей в контексте временных автоматных моделей. В частности, была построена категория временных систем переходов, доказаны ее основные свойства, например коуниверсальность, и получена теоретико-категорная характеризация временного варианта тестовой эквивалентности. В дальнейшем планируется расширить полученный результат на другие варианты временных эквивалентностей, например на временные варианты слабых эквивалентностей.

СПИСОК ЛИТЕРАТУРЫ

1. **Цаленко М.Ш., Шульгейфер Е.Г.** Лекции по теории категорий. — М: Наука, 1974. — 438 с.
2. **Грибовская Н.С.** Теоретико-категорная характеризация различных эквивалентностей на временных автоматных моделях. — Новосибирск, 2004. — 38 с. — (Препр. / СО РАН Ин-т систем информатики; № 119).
3. **Alur R., Dill D.L.** A theory of timed automata // Theor. Comput. Sci. — 1994. — Vol. 126.
4. **Hoare C.A.R.** Communicating Sequential Processes. — Prentice-Hall, 1985.
5. **Hune T., Nielsen M.** Timed bisimulation and open maps // Lect. Notes Comput. Sci. — Berlin etc., 1998. — Vol. 1450. — P. 378–387.
6. **Hune T., Nielsen M.** Timed bisimulation and open maps. — BRICS, 1998. — (Tech. Rep. / Department of Computer Science / University of Aarhus; RS-98-4).
7. **Joyal A., Nielsen M., Winskel G.** Bisimulation from open maps // Information and Computation. — 1996. — Vol. 127(2). — P. 164–185.
8. **Nielsen M., Cheng A.** Observing behaviour categorically // Lect. Notes Comput. Sci. — Berlin etc., 1996. — Vol. 1026. — P. 263–278.
9. **Virbitskaite I.B., Gribovskaya N.S.** Open Maps and Observational Equivalences for Timed Partial Order Models // Fundamenta Informaticae. — 2004. — Vol. 61. — P. 383–399.

А. В. Демин, Е. Е. Витяев

РАЗРАБОТКА МОДЕЛИ АДАПТИВНОГО ПОВЕДЕНИЯ АНИМАТА НА ОСНОВЕ СЕМАНТИЧЕСКОГО ВЕРОЯТНОСТНОГО ВЫВОДА

1. ВВЕДЕНИЕ

В последнее время активно развивается направление исследований «Адаптивное поведение», связанное с изучением фундаментальных принципов, позволяющих естественным или искусственным организмам приспосабливаться к переменной внешней среде. Один из основных подходов этого направления является создание и исследование агентов (компьютерных программ или роботов), поведение которых основано на принципах поведения живых организмов. Подобные агенты были названы «аниматами» (animal + automat = animat).

В данной работе предложена общая схема адаптивной системы управления аниматом, которая включает в себя архитектуру на основе иерархии функциональных систем и подцелей, алгоритм обучения, использующий семантический вероятностный вывод и возможность автоматического формирования новых подцелей. На основе предложенной модели в виде компьютерной программы был реализован простейший анимат и среда его обитания. При помощи данной программы был поставлен ряд экспериментов по обучению анимата и проведено тестовое сравнение с существующими подходами, основанными на нейронных сетях и потактовом обучении (Reinforcement Learning).

2. ТЕОРИЯ ФУНКЦИОНАЛЬНЫХ СИСТЕМ

Архитектура предложенной нами системы управления основана на теории функциональных систем, разработанной в 1930–1970 гг. известным русским нейрофизиологом П.К. Анохиным [9]. Согласно этой теории единицей деятельности организма является функциональная система, формирующаяся для достижения полезных для организма результатов (например, удовлетворение потребностей). Организация функциональных систем при целенаправленном поведении осуществляется в соответствии с двумя правилами: последовательностью и иерархией результатов. Последователь-

ность результатов выстраивается по принципу “доминанты”: доминирующая потребность возбуждает доминирующую функциональную систему и строит поведенческий акт, направленный на ее удовлетворение. По отношению к доминирующей функциональной системе все остальные функциональные системы выстраиваются в иерархию по принципу “иерархии результатов”: когда результат деятельности одной функциональной системы входит в качестве компонента в результат деятельности другой.

Центральные механизмы функциональных систем, обеспечивающих целенаправленные поведенческие акты, имеют однотипную архитектуру. Начальную стадию поведенческого акта любой степени сложности составляет *афферентный синтез*, включающий в себя синтез мотивационного возбуждения, памяти и информации об окружающей среде. В результате афферентного синтеза из памяти извлекаются все возможные способы достижения цели в данной ситуации. На стадии *принятия решений* в соответствии с исходной потребностью выбирается только один конкретный способ действий. Для обеспечения достижения результата еще перед началом действий формируется *акцептор результатов действий*, представляющий собой модель параметров ожидаемого результата. Выполнение каждого действия постоянно сопровождается сигналом о получении результата, называемой *обратной афферентацией*. Действия по достижению цели продолжаются до тех пор, пока параметры результата действия, поступающие в центральную нервную систему в форме соответствующей обратной афферентации, не будут полностью соответствовать свойствам акцептора результатов действия.

Отдельная ветвь общей теории функциональных систем — теория системогенеза, изучающая закономерности формирования функциональных систем. В данной работе мы также рассмотрим механизм формирования новых функциональных систем на основе выявления подцелей.

3. МОДЕЛЬ АНИМАТА

Приведем схему работы анимата (рис.1), реализующую схему функциональных систем [1—5]. Будем предполагать, что система управления аниматом функционирует в дискретном времени $\boldsymbol{t} = \boldsymbol{0}, \boldsymbol{1}, \boldsymbol{2}, \ldots$. Пусть анимат имеет некоторый набор сенсоров $\boldsymbol{S_1}, \boldsymbol{S_2}, \ldots, \boldsymbol{S_n}$, характеризующих состояние внешней и внутренней среды, и набор возможных действий $\boldsymbol{A_1}, \boldsymbol{A_2}, \ldots, \boldsymbol{A_m}$. Среди множества сенсоров выделим сенсор $\boldsymbol{SA}$, который представляет информацию о совершенном действии. Считаем, что история деятельности

анимата хранится в таблице данных $X = \{X_1, ..., X_t\}$, где *t*-я строка таблицы содержит показания сенсоров в момент времени *t*: $X_t = \{S_1^t, S_2^t, ..., S_n^t, SA_t\}$, где $S_1^t, S_2^t, ..., S_n^t$ — значения сенсоров $S_1, S_2, ..., S_n$ в момент времени *t*. На множестве X определим множество предикатов $P = \{P_1(t), ..., P_k(t), PA_1(t), ..., PA_m(t)\}$, где $P_i(t)$ — *сенсорные предикаты*, определяющие некоторые условия на показания сенсоров в момент времени *t*; $PA_i(t) \Leftrightarrow (SA(t) = A_i)$ — *активирующие предикаты*, показывающие, что в момент времени *t* было совершено действие A_i.

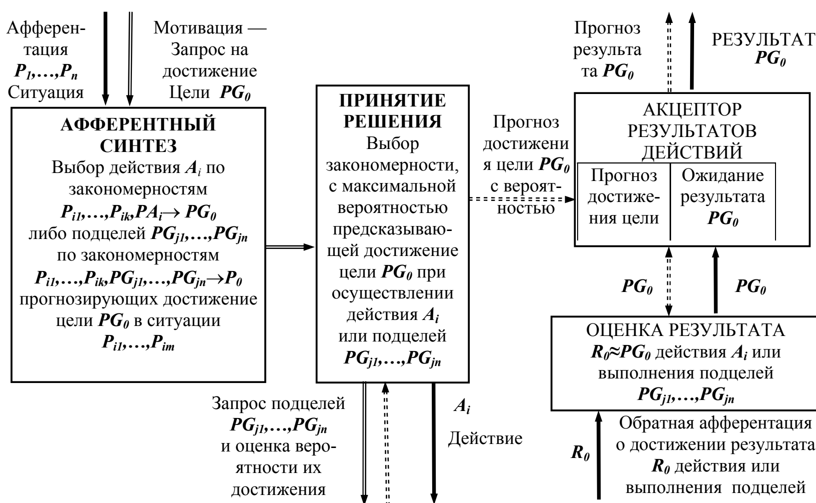


Рис. 1. Схема работы функциональной системы

Введем понятие *предиката-цели* — $PG(t) = P_{i1}(t) \,\&\, P_{i2}(t) \,\&\, ... \,\&\, P_{il}(t)$, реализующего условие достижения цели в момент времени *t*.

Каждой функциональной системе $ФC_j$ соответствует некоторая цель G_j, достижение которой является задачей данной функциональной системы, и предикат-цель PG_j, характеризующий условие достижения цели.

Каждая функциональная система $\boldsymbol{\Phi C_j}$ содержит свой набор предикатов $\boldsymbol{P_j = P \cup \{PG_{j1}, ..., PG_{jn}\}}$, где $\boldsymbol{PG_{jk}}$ — предикаты-цели, соответствующие целям нижестоящих по иерархии функциональных систем, подчиненных данной функциональной системе. Каждая функциональная система $\boldsymbol{\Phi C_j}$ содержит множество закономерностей $\boldsymbol{PR_j}$ вида: $\boldsymbol{P_{i1}, ..., P_{ik}, PG_{j1}, ..., PG_{jn}, PA_i \to PG_j}$. Каждая такая закономерность характеризуется некоторой оценкой $\boldsymbol{p}$ вероятности достижения цели $\boldsymbol{PG_j}$, при выполнении условия закономерности.

Предположим, что в некоторый момент времени *t* функциональная система $\boldsymbol{\Phi C_j}$ получила запрос на достижение цели $\boldsymbol{PG_j}$. Тогда из множества закономерностей $\boldsymbol{PR_j}$ извлекаются все закономерности, условие которых выполнено в текущий момент времени *t*. Если условие закономерности содержит предикаты-подцели $\boldsymbol{PG_{j1}, ..., PG_{jn}}$, то функциональная система отправляет запрос на достижение этих подцелей вниз по иерархии функциональных систем. Среди всех отобранных закономерностей выбирается та закономерность, которая с учетом вероятностей выполнения подцелей дает максимальную оценку *f* вероятности достижения цели. Оценка *f* закономерности $\boldsymbol{P_{i1}, ..., P_{ik}, PG_{j1}, ..., PG_{jn}, PA_i \to PG_j}$ вычисляется следующим образом: $\boldsymbol{f(PG_j \mid PS_{i1}, ..., PS_{ik}, PG_{j1}, ..., PG_{jn}, PA_i) = p \cdot f(PG_{j1}) \cdot ... \cdot f(PG_{jn})}$, где *p* — оценка вероятности данной закономерности, $\boldsymbol{f(PG_{jk})}$ — оценка вероятностей достижения подцелей.

Если все условия выбранной закономерности выполнены, то действие $\boldsymbol{A_i}$ запускается на выполнение. Если множество закономерностей $\boldsymbol{PR_j}$ пусто либо нет ни одной закономерности, применимой в данной ситуации, то действие выбирается случайно из арсенала имеющихся действий.

После совершения действия обновляются показания сенсоров, оценивается результат действия и уточняется набор правил $\boldsymbol{PR_j}$ (см. ниже).

4. ОЦЕНКА РЕЗУЛЬТАТОВ ДЕЙСТВИЙ

Каждая функциональная система $\boldsymbol{\Phi C_j}$ хранит оценки результатов своих действий $\boldsymbol{d^j(t)}$ для каждого момента времени *t*. Определим способ оценки результатов действий.

Предположим, что функциональной системе $\boldsymbol{\Phi C_j}$ в момент времени $\boldsymbol{t_0}$ была поставлена задача достичь цель $\boldsymbol{G_j}$, и после достижения цели в момент времени $\boldsymbol{t_1}$ был получен результат $\boldsymbol{R_j}$. Тогда оценки результатов действий $\boldsymbol{d^j(t)}$, начиная с момента времени $\boldsymbol{t_0}$ и до момента времени $\boldsymbol{t_1}$, будут рассчитаны следующим образом:

$$d^j(t) = r\frac{t - t_0}{(t_1 - 1) - t_0}, t_0 < t < t_1,$$

где $\boldsymbol{r}$ — функция оценки качества полученного результата,

$$r = \begin{cases} 0, \text{ если } PG_j = 0 \\ \| G - R \|, \text{ если } PG_j = 1 \end{cases},$$

где ‖…‖ — мера близости между полученным результатом $\boldsymbol{R_j}$ и поставленной целью $\boldsymbol{G_j}$.

5. ГЕНЕРАЦИЯ ПРАВИЛ

Для получения множества закономерностей $\boldsymbol{PR_j}$, которые использует функциональная система $\boldsymbol{\Phi C_j}$, воспользуемся семантическим вероятностным выводом [6].

Семантический вероятностный вывод позволяет находить все закономерности вида $\boldsymbol{P_{i1}, ..., P_{in} \to P_0}$, с максимальной вероятностью предсказывающие предикат $\boldsymbol{P_0}$. Вывод осуществляется на некотором множестве обучающих данных **Y** с использованием заданного множества предикатов $\{\boldsymbol{P_1, ..., P_m}\}$.

Данный метод основывается на следующем определении вероятностной закономерности, предложенном в работе [7].

Правило $\boldsymbol{P_{i1}, ..., P_{in} \to P_0}$ является *закономерностью*, если оно удовлетворяет следующим условиям.

1) $p(P_{i1},...,P_{in}) > 0$,

2) $\forall \{P_{ij},...,P_{ik}\} \subset \{P_{i1},...,P_{in}\} \quad p(P_0 \mid P_{i1},...,P_{in}) > p(P_0 \mid P_{ij},...,P_{ik})$.

Здесь p — оценка условной вероятности правила.

Введем понятие *уточнения* правила. Правило $P_{i1},...,P_{in},P_{in+1} \to P_0$ является *уточнением* правила $P_{i1},...,P_{in} \to P_0$, если оно получено добавлением в посылку правила $P_{i1},...,P_{in} \to P_0$ произвольного предиката P_{in+1}, и $p(P_0 \mid P_{i1},...,P_{in+1}) > p(P_0 \mid P_{i1},...,P_{in})$.

Алгоритм семантического вероятностного вывода.

- На первом шаге генерируется множество уточнений правила $\to P_0$ (т.е. правила с пустой посылкой). Это множество будет состоять из правил единичной длины, имеющих вид $P_{ij} \to P_0$, для которых $p(P_0 \mid P_{ij}) > p(P_0)$.
- На k-м (k > 1) шаге генерируется множество уточнений всех правил, созданных на предыдущем шаге. Т.е. для каждого правила $P_{i1},...,P_{ik-1} \to P_0$, сгенерированного на (k-1)-м шаге, создается множество правил вида $P_{i1},...,P_{ik-1},P_{ik} \to P_0$, таких что $p(P_0 \mid P_{i1},...,P_{ik-1},P_{ik}) > p(P_0 \mid P_{i1},...,P_{ik-1})$.
- Проверяется, являются ли полученные правила закономерностями. Правила, не удовлетворяющие условиям закономерности, отсеиваются.
- Алгоритм останавливается, когда больше невозможно уточнить ни одно правило.

Для того чтобы избежать генерации статистически незначимых правил, вводиться дополнительный критерий — оценка на статистическую значимость. Правила, не удовлетворяющие этому критерию, отсеиваются, даже если они имеет высокую точность на обучающем множестве. Для оценки статистической значимости в алгоритме используется критерий Фишера (точный критерий Фишера для таблиц сопряженности) [8].

Очевидно, что все правила, полученные при помощи данного алгоритма, будут являться закономерностями. На рис. 2 представлено дерево вывода, соответствующее описанному процессу.

Чтобы найти все закономерности $P_{i1},...,P_{ik},PG_{j1},...,PG_{jn},PA_i \to PG_j$, с максимальной вероятностью предсказывающие достижение цели G_j, стро-

иться дерево семантического вероятностного вывода на множестве данных истории деятельности анимата **X** и множестве оценок действий $\boldsymbol{d}^j(\boldsymbol{t})$ с использованием набора предикатов $\boldsymbol{P}_j$, которые использует данная функциональная система. Оценка условной вероятности $\boldsymbol{p}$ правила рассчитывается следующим образом: $\boldsymbol{p} = \sum_{i \in I} \boldsymbol{d}_i^j \Big/ \| \boldsymbol{I} \|$, где I — множество моментов времени, когда может быть применено данное правило.

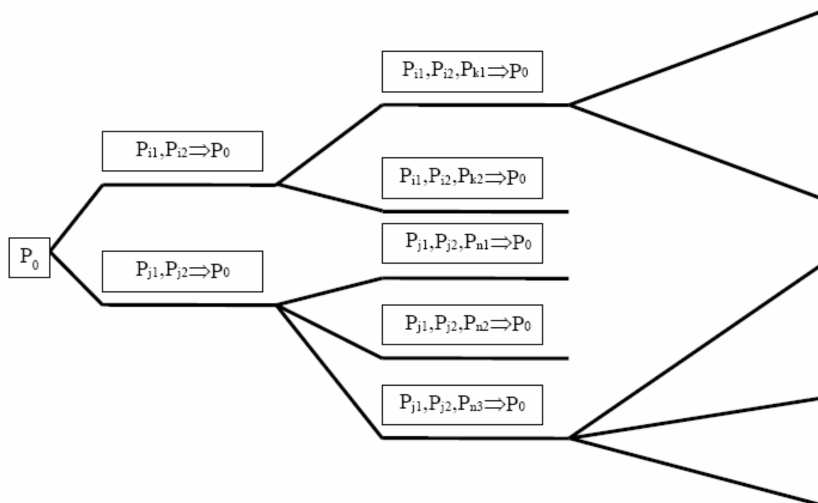


Рис. 2. Дерево семантического вероятностного вывода

6. ИЗВЛЕЧЕНИЕ ПОДЦЕЛЕЙ

Изначально система управления аниматом имеет заданную априори иерархию функциональных систем. В простейшем случае она может состоять всего из одной функциональной системы. В процессе деятельности система управления может автоматически выявлять новые подцели и порождать соответствующие функциональные системы. Опишем процедуру порождения новых подцелей и функциональных систем.

Определим подцель как ситуацию, достижение которой значительно увеличивает вероятность достижения вышестоящей цели, и последующие действия из этой ситуации не могут быть определены однозначно.

Для выявления подцелей анализируется множество правил PR_j функциональной системы. Ситуация, описываемая предикатом $PG_i = P_1 \& \ldots \& P_k$, будет являться подцелью G_i, если выполнены следующие условия:

1) для любого правила $R_1 = P_{i1},\ldots,P_{in},PA_i \to PG_j$ такого, что $\{P_1,\ldots,P_k\} \subseteq \{P_{i1},\ldots,P_{in}\}$, и для любого $R_2 = P_{j1},\ldots,P_{jm},PA_j \to PG_j$ такого, что $\{P_{j1},\ldots,P_{jm}\} \subset \{P_{i1},\ldots,P_{in}\}$ и $\{P_1,\ldots,P_k\} \not\subset \{P_{j1},\ldots,P_{jm}\}$, выполнено условие $p(R_1) - p(R_2) > \delta$;
2) существуют правила $R_1 = P_{i1},\ldots,P_{in},PA_i \to PG_j$ и $R_2 = P_{j1},\ldots,P_{jm},PA_j \to PG_j$ такие, что $\{P_1,\ldots,P_k\} \subseteq \{P_{i1},\ldots,P_{in}\}$, $\{P_1,\ldots,P_k\} \subseteq \{P_{j1},\ldots,P_{jm}\}$ и $A_i \neq A_j$.

Первое условие говорит о том, что добавление данной ситуации в условную часть правил должно значительно увеличивать оценку условной вероятности правил (более чем на δ, где δ — некоторый порог, например δ = 0.2), это означает, что достижение такой ситуации значительно увеличивает вероятность достижения вышестоящей цели. Второе условие говорит о том, что после достижения данной ситуации возможны различные дальнейшие действия.

Таким образом, у каждой функциональной системы $ФС_j$ анализируется множество ее правил PR_j и выявляются новые подцели. Для каждой обнаруженной подцели G_i создается новая функциональная система $ФС_i$, находящаяся ниже по иерархии системы $ФС_j$ и реализующая эту подцель. Для созданной функциональной системы $ФС_i$ при помощи семантического вероятностного вывода порождается множество закономерностей PR_i. Для этого просматривается все множество данных истории анимата X и выявляются случаи, когда подцель G_i была реализована и рассчитывается множество оценок действий $d^i(t)$ функциональной системы $ФС_i$ описанным выше способом. Для всех функциональных систем, находящихся на один уровень выше $ФС_i$, набор предикатов обогащается еще одним предикатом PG_i и генерируются новые правила. Тем самым, множество закономерно-

стей этих функциональных систем обогащаются закономерностями, содержащими новую подцель $\boldsymbol{G}_i$.

7. ОПИСАНИЕ ЭКСПЕРИМЕНТА

Для исследования описанной выше системы управления был поставлен следующий эксперимент. При помощи компьютерной программы был смоделирован виртуальный мир и функционирующий в нем анимат, целью которого является сбор специальных объектов виртуального мира — «еды».

Мир анимата представляет собой прямоугольное поле, разбитое на клетки, и содержит четыре типа объектов: пустые клетки («трава»), препятствия («препятствие»), еду («еда») и таблетки («таблетка»). Объекты «препятствие» располагаются только по периметру виртуального мира, образуя тем самым его естественные границы. Анимат может перемещаться по полю, совершая три типа действий: шаг на клетку вперед («шаг»), поворот налево («налево»), поворот направо («направо»).

Данный эксперимент является усложнением известной тестовой поведенческой задачи фуражирования, в которой анимат должен научиться эффективно находить и собирать пищевые объекты. В нашем эксперименте виртуальный мир содержит еще один объект, который мы условно назвали «таблетка». Чтобы съесть еду, анимат должен иметь при себе таблетку, которую он должен предварительно найти и подобрать. Когда он съест еду, таблетка исчезнет, и, чтобы съесть следующую еду, он должен опять найти и подобрать таблетку, и т.д.

Чтобы съесть еду или подобрать таблетку, анимату достаточно шагнуть на клетку, содержащую соответствующий объект. Однако если у него нет таблетки, то он не сможет съесть еду, и если анимат уже имеет одну таблетку, то пока он ее не использует для поедания еды, он больше не сможет подобрать ни одной таблетки. Изначально таблетки и еда случайным образом располагаются по полю. Когда анимат съедает еду или подбирает таблетку, то клетка, на которой находился этот объект, очищается, и новый объект такого же типа появляется в случайном месте поля, т.е. количество еды и таблеток в виртуальном мире всегда поддерживается постоянным.

Для ориентации в виртуальном мире анимат имеет десять сенсоров: «объект впереди-слева», «объект впереди», «объект впереди-справа», «объект слева», «объект в центре», «объект справа», «объект сзади-слева», «объект сзади», «объект сзади-справа» и «есть таблетка». Первые девять сенсоров, в соответствии со своими названиями, информируют анимат о типах

объектов, расположенных в соответствующих клетках, и принимают значения «трава», «препятствие», «еда» или «таблетка». Например, сенсор «объект впереди» информирует о состоянии клетки перед аниматом, сенсор «объект в центре» — о состоянии клетки, на которой находиться анимат и т.д. Еще один сенсор «есть таблетка» информирует анимат о наличие таблетки и принимает значения «да» или «нет».

Изначальный набор предикатов анимата состоит из тридцати семи сенсорных предикатов: по четыре предиката на каждый сенсор s, информирующий анимат о состоянии окружающих его клеток: **(s = «трава»)**, **(s = «препятствие»)**, **(s = «еда»)**, **(s = «таблетка»)**, и один предикат, говорящий о наличие таблетки: **(«есть таблетка» = «да»)**. А также трех активирующих предикатов: **(A = «шаг»)**, **(A = «налево»)** и **(A = «направо»)**.

При старте система управления аниматом состоит только из одной, базовой, функциональной системы, целью которой является достижение ситуации одновременного наличия таблетки и ощущения еды центральным сенсором, соответствующий предикат-цель имеет вид $\boldsymbol{PG_0}$ = **(«центр» = «еда» И «есть таблетка» = «да»)**. Когда анимат достигает эту цель, он «поедает» еду.

8. РЕЗУЛЬТАТЫ ЭКСПЕРИМЕНТА

Одной из основных задач эксперимента является демонстрация возможности автоматического формирования иерархии целей и результатов в целенаправленном поведении. В ходе эксперимента система управления аниматом при каждом тестовом запуске стабильно обнаруживала новую подцель, описываемую предикатом-целью $\boldsymbol{PG_1}$ **= («есть таблетка» = «да»)**, и создавала для нее соответствующую функциональную систему. Работа системы управления происходила следующим образом. При отсутствии у анимата таблетки срабатывала закономерность $\boldsymbol{PG_1 \rightarrow PG_0}$ как наиболее вероятная в данной ситуации, которая передавала управление нижестоящей функциональной системе, реализующей поиск таблетки. Когда таблетка найдена у базовой функциональной системы начинали срабатывать правила с более высокой вероятностью, в результате чего она находила еду.

Для того чтобы оценить эффективность предлагаемой нами системы управления, в экспериментах также проводилось тестовое сравнение с системами, построенными на основании теории обучения с подкреплением (Reinforcement Learning), описанной в работах Р. Саттона и Э. Барто [10].

Для сравнения мы выбрали две системы управления, построенные на основе популярного алгоритма обучения с подкреплением Q-Learning. Суть алгоритма заключается в последовательном уточнении оценок суммарной величины награды $Q(s_t, A_t)$, которую получит система, если в ситуации s_t она выполнит действие A_t, по формуле:

$$Q^{(i+1)}(s_t, A_t) = Q^{(i)}(s_t, A_t) + \alpha(r_t + \gamma \max_A Q^{(i)}(s_{t+1}, A) - Q^{(i)}(s_t, A_t)).$$

Первая из этих двух систем (Q-Lookup Table) основана на использовании таблицы, которая содержит Q-значения для всех возможных ситуаций и действий. Изначально эти значения таблицы заполняются случайным образом. В процессе работы в каждый такт времени система совершает действие и уточняет соответствующие Q-значения.

Вторая система (Q-Neural Net) использует аппроксимацию функции $Q(s_t, A_t)$ при помощи нейронных сетей. При этом для каждого возможного действия A_i используется своя нейронная сеть NN_i. В каждый такт времени система выбирает действие, чья нейронная сеть выдаст наибольшую оценку Q-значения, после чего действие совершается и происходит адаптация весов соответствующей нейронной сети.

Тестовое сравнение проводилось на поле размером 25 на 25 клеток. Количество таблеток и еды на поле поддерживалось постоянным: по 100 объектов каждого типа. Весь период функционирования анимата был разбит на этапы по 1000 шагов (тактов). Оценивалось, какое количество еды соберет анимат с разными системами управления за каждый этап работы. Очевидно, что после того как система управления полностью обучится и достигнет своего оптимального поведения, анимат начнет собирать примерно одно и то же количество еды за один этап. Таким образом, можно оценить как эффективность каждой системы управления в целом, так и скорость ее обучения.

На рис. 3 представлены результаты тестового сравнения. Для каждой системы управления рассчитывались средние значения по результатам 20-ти испытаний. Продолжительность каждого испытания составляла 100 000 шагов, за это время анимат должен был научиться эффективно решать поставленную задачу. Как видно на графике, система управления на основе семантического вероятностного вывода превосходит системы Reinforcement Learning как по скорости обучения, так и по качеству работы.

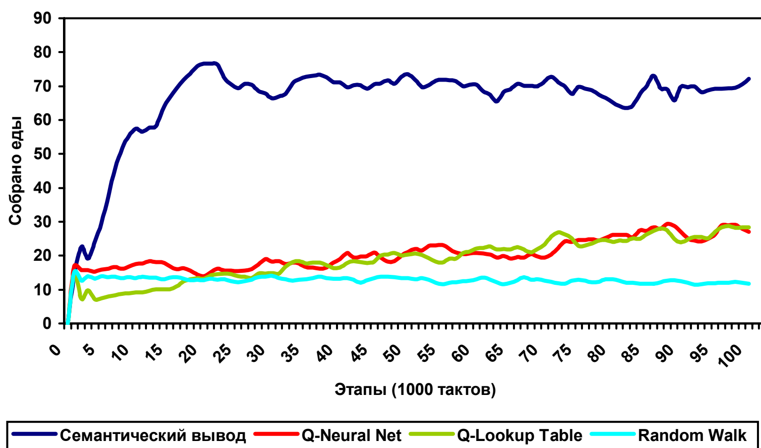


Рис. 3. Количество «еды», собранное аниматом с разными системами управления

Системы управления на основе Reinforcement Learning в данном эксперименте показали плохую обучаемость и нестабильную работу. Основная проблема в работе этих систем была связана с тем, что они не могли за приемлемое время научиться стабильно адекватно реагировать на показания сенсоров о наличие таблеток и зачастую проходили мимо таблеток даже после 100 000 шагов обучения.

Дополнительные эксперименты показали, что система управления на основе нейронных сетей (Q-Neural Net) при увеличении длительности обучения до 300 000–500 000 шагов в некоторых случаях способна обучиться правильно реагировать на все показания сенсоров. Однако, по нашему мнению, столь длительный срок обучения является неприемлемым для адаптивной системы.

Система управления на основе использования таблицы Q-значений не смогла достичь оптимального поведения даже после 500 000 шагов. Во многом это связано с большим количеством возможных ситуаций: в данной задаче анимат может столкнуться с 137 538 различными ситуациями.

9. ВЫВОДЫ

Таким образом, результаты эксперимента показывают, что в условиях усложнения среды умение формировать и достигать подцели является принципиальным для эффективного достижения конечных целей. Несмотря на то что в данной модели адаптивной системы управления используется достаточно простой способ формирования подцелей, уже эта возможность дает значительные преимущества в обучении. Как видно из эксперимента, использование иерархии функциональных систем и алгоритма выявления подцелей позволяет предлагаемой нами системе управления эффективно обучаться и решать поставленную задачу. Существующие подходы, основанные на нейронных сетях и Reinforcement Learning, не могут автоматически выявлять подцели и поэтому значительно проигрывают в усложненных экспериментах.

СПИСОК ЛИТЕРАТУРЫ

1. Витяев Е.Е. Целеполагание как принцип работы мозга // Вычислительные системы. — 1997. — № 158. — С. 9–52.
2. Витяев Е.Е. Вероятностное прогнозирование и предсказание как принцип работы мозга // Вычислительные системы. — 1998. — № 162. — С. 14–40.
3. Витяев Е.Е. Формальная модель работы мозга, основанная на принципе предсказания // Вычислительные системы. — 1998. — № 164. — С. 3–61.
4. Михиенко Е.В., Витяев Е.Е. Моделирование работы функциональной системы // Тр. VI Всерос. научно-технической конф. «Нейроинформатика-2004». — М., 2004. — Ч. 2. — С. 124–129.
5. Витяев Е.Е. Объяснение Теории Движений Н.А.Бернштейна. Тр. VII Всерос. научно-технической конф. «Нейроинформатика-2005». — М., 2005. — Ч. 1. — С. 234–240.
6. Витяев Е.Е. Семантический подход к созданию баз знаний. Семантический вероятностный вывод // Вычислительные системы. — 1992. — № 146. — С. 19–49.
7. Витяев Е.Е. Метод обнаружения закономерностей и метод предсказания // Вычислительные системы. — 1976. — № 67. — С. 54–68.
8. Кендал М., Стюарт А. Статистические выводы и связи. — М.: Наука, 1973. — 899 с.
9. Анохин П.К. Принципиальные вопросы общей теории функциональных систем // Принципы системной организации функций. — М.: Наука, 1973. — См. также: http://www.keldysh.ru/pages/BioCyber/RT/Functional.pdf
10. Sutton R., Barto A. Reinforcement Learning: An Introduction. — Cambridge: MIT Press, 1998. — See also: http://www-anw.cs.umass.edu/~rich/book/the-book.html

В.В. Кальченко

XML-АЛГЕБРА ДЛЯ ЯЗЫКА ЗАПРОСОВ XQUERY[1]

1. ВВЕДЕНИЕ

В последнее время требования к базам данных значительно возросли, и стало невозможно решать все задачи с помощью реляционных баз данных без значительного усложнения внутренней структуры данных. В качестве альтернативы рассматриваются XML-базы данных, предоставляющие большие возможности в организации данных. До сих пор не существует всеми признанного стандарта для языка запросов, такого как SQL для реляционных баз. Наиболее перспективным в этом смысле языком является XQuery [5]. Разработка алгебры для XQuery — первоочередная задача при создании XML-базы данных. Для XQuery разработано множество алгебр [3, 4, 5, 6, 7, 9, 15, 16, 17]. Типичные ошибки, совершаемые авторами алгебр:

- использование искусственной модели данных, придуманной самими авторами;
- неформальное описание операций, при котором опускаются важные детали;
- представление понятия “алгебра” отличным от данного понятия в математике;
- игнорирование того, что результат операции может не принадлежать той же алгебре, что и аргументы.

В данной работе рассматривается алгебра [17], свободная от приведенных выше недостатков. В работе [10] дано формальное определение модели XML-базы данных, соответствующей модели данных XQuery 1.0 и XPath 2.0 [13] и состоящей из деревьев документов, определенных схемой XML Schema [21]. Рассматриваемая алгебра представлена в виде многоосновной алгебры [12] и использует эту модель данных. Алгебра будет расширена пространствами имен [14].

Статья организована следующим образом: во второй части даны основные понятия языка XQuery и структура XML-документов, трятья часть содержит краткий обзор алгебр, разработанных для XQuery, в четвертой части описана алгебра [17], расширенная пространствами имен.

[1] Работа выполнена при поддержке РФФИ, грант № 04-01-00272.

2. ОСНОВНЫЕ ПОНЯТИЯ

XML-документ [20] состоит из *информационных единиц* с определенным на них *документным порядком*. В этом порядке они следуют в текстовой версии документа. Информационные единицы хранятся в базе данных в виде *узлов*. XML-документ можно представить в виде упорядоченного дерева, корнем которого является документный узел. Каждый элемент может иметь дочерние элементы и атрибуты (соединен с ними дугами). В модели данных XQuey [13] каждый узел имеет идентификатор и состояние, данные о которых можно получить с помощью *аксессоров* (например*, node_kind, node_name, base_uri, type_name, type_value, attributes, parent, children* и т.д.). Каждый узел принадлежит типу Node, который является объединением типов Document, Element, Attribute, Text, Namespace, ProcInstruction и Comment. Каждый узел и каждое атомарное значение принадлежат также типу Item, который, в свою очередь, является объединением типов xdt:anyAtomicType (объединение всех атомарных типов и xdt:untypedAtomic) и Node. Все узлы линейно упорядочены таким образом, что если какой-то узел одного документа предшествует узлу другого документа, то все узлы первого документа предшествуют данному узлу второго документа. Структура XML-документа обычно задается XML-схемой [21]. В схеме определяются типы элементов и атрибутов, отношения "дочки-матери" между элементами, пользовательские типы данных и т.д.

Схема XML-документа может разрабатываться разными авторами и содержать в себе одинаковые имена для различных элементов и атрибутов. Вследствие этого возможны коллизии имен, и необходимо решать проблемы идентификации именованных объектов. Во избежание коллизий в XML введены понятия пространства имен (namespace) и расширенных имен (expanded names). Расширенное имя — это пара, состоящая из имени пространства имен и локального имени. При этом имя пространства имен определяется с помощью ссылки URI[2] [11] с некоторыми ограничениями, а именно — имя пространства имен не может быть пустым и документ не может содержать взаимозависимые ссылки. Ссылки URI могут содержать недопустимые символы и быть очень длинными, поэтому расширенные имена не используются напрямую для задания имен элементов и атрибутов, вместо них используются уточненные имена (qualified names), которые состоят из локального имени и префикса, определяющего имя пространства

[2] Обычно ссылка URI представлена в виде строки, содержащей в себе адрес схемы (в Интернете, на локальном диске и т.д.), определяющей данное пространство имен, например: "http://www.w3.org/XML/1998/namespace"

имен. Префикс может быть пустым, что означает принадлежность данных к пространству имен, указанному для использования по умолчанию.

Язык XQuery используется для написания запросов к XML-базам данных. Обычно запрос выглядит следующим образом:

for x_1 **in** s_1, x_2 **in** $s_2(x_1)$, ..., x_m **in** $s_m(x_1,...,x_{m-1})$
let $y_1 := e_1(x_1,...,x_m)$, $y_2 := e_2(x_1,...,x_m,y_1)$,..., $y_n := e_n(x_1,...,x_m,y_1,...,y_n)$
where $p(x_1,...,x_m,y_1,...,y_n)$
order by $e(x_1,...,x_m,y_1,...,y_n)$
return $f(x_1,...,x_m,y_1,...,y_n)$,

где s_i — последовательности, x_i, y_i — переменные, а p, e и f — выражения. Операция **for** определяет область, откуда берутся значения, **let** позволяет вычислить промежуточные значения, необходимые для остальных операций, **where** задает условие на аргументы, **order by** задает порядок полученных в результате вычисления значений и **return** — структуру возвращаемого XML-фрагмента. Такой запрос обычно называют выражением FLWOR. В более ранних версиях XQuery не было операции **order by**, и запрос назывался FLWR. В выражениях XQuery широко используются путевые выражения [18], которые позволяют задать путь в дереве XML-документа. Путевое выражение обычно содержит в себе последовательность узлов (следующих в порядке вложенности), указывающую на то, как достигнуть последнего узла последовательности, начиная с первого. Выражения XQuery, помимо вложенных FLWOR выражений и путевых выражений, могут содержать в себе выражения создания и управления последовательностями (такие как объединение, пересечение двух последовательностей и т.д.) и кванторы существования и универсальности.

При обработке запросов к базе данных выражение на языке запроса обычно транслируется в выражение соответствующей алгебры, и затем производится вычисление полученного выражения. Алгебра используется по двум основным причинам. Во-первых, алгебра формально определяет семантику языка запросов. Во-вторых, алгебра обычно поддерживает набор правил оптимизации полученного в результате трансляции выражения. При разработке алгебр для XML-данных возникают три ключевых проблемы: вычисление путевых выражений, вычисление и выполнение вложенных запросов, сохранение документного порядка при вычислениях.

Алгебру для XQuery удобно рассматривать как многоосновную алгебру [12]. Многоосновная сигнатура Σ определяется как пара (T, F), где T —

множество основ, а F — множество символов операций, каждому из которых сопоставлен профиль. Символ операции — это символ или имя, а профиль — это элемент T (в этом случае профиль определяет константу) или $t_1,\ldots,t_n \rightarrow t_{n+1}$ (в этом случае профиль определяет функцию), где t_i — элементы T.

Многоосновная алгебра A сигнатуры Σ содержит в себе:

- множество A_t для каждого элемента $t \in T$,
- элемент $c^A \in A_t$ для каждой операции c с профилем t,
- функцию $f^A : A_{t1} \times \ldots \times A_{tn} \rightarrow A_{tn+1}$ для каждого символа операции f с профилем $t_1,\ldots,t_n \rightarrow t_{n+1}$.

Набор множеств алгебры A называется *носителем алгебры* и обозначается $|A|$. Алгебра сигнатуры Σ называется Σ-алгеброй. Множество T состоит из множества имен типов, а множество F — из операций, определенных на этих типах.

3. КРАТКИЙ ОБЗОР СУЩЕСТВУЮЩИХ АЛГЕБР

В обзоре [10] рассмотрены работы по алгебрам для XQuery. Можно выделить два основных подхода при построении алгебры. Первый заключается в расширении возможностей реляционных алгебр, путем добавления необходимых операций для работы с XML, например, работы [9, 16]. При использовании второго подхода алгебра строится с нуля, как сделано в работах [3, 17]. В обоих случаях есть как преимущества, так и недостатки. При построении алгебры на основе реляционной алгебры можно использовать уже имеющиеся правила оптимизации выражений и сохраняется возможность работать не только с XML-данными, но и с данными в реляционном формате. Минусы этого подхода тоже очевидны — недостаточная гибкость алгебры не позволяет использовать все преимущества XML без специальных ухищрений. При втором подходе обычно используется один из двух видов моделей данных: сложная модель с деревьями в качестве информационных единиц [3, 5] и простая модель, основанная на модели, предложенной WC3 [13].

Несколько алгебр было разработано в процессе проектирования базы данных TIMBER [2]. Алгебра TAX, использующая деревья, описана в работе [3]. База данных в этой алгебре состоит из коллекции помеченных деревьев, все операции алгебры используют в качестве аргументов коллекции деревьев и возвращают коллекции деревьев. Сложная модель данных принудила авторов к разработке физической алгебры [4], в которой уже ис-

пользуется достаточно простая модель данных. В дальнейшем авторы используют немного модифицированную физическую алгебру, как основу для новой алгебры [5], в которой используются *обобщенные шаблоны дерева* (GTP). С помощью GTP запрос XQuery можно представить в виде одного или нескольких деревьев. Все операции этой алгебры описаны неформально. На следующем шаге в разработке этого проекта была представлена еще одна алгебра [6], использующая понятие *логических классов*. Логический класс — помеченное множество узлов дерева, соответствующих определенному узлу в шаблоне. Для приведения разнородных деревьев к одному виду используется понятие *сужения логического класса*. Этим решается проблема применения операций, рассчитаных на работу с множествами однотипных элементов. Формальное определение операций отсутствует.

Алгебра, чьи операции работают в заданном контексте, описана в статье [7]. Контекст представляет собой позицию в документе, которая для операций работает как указатель на аргументы. В качестве аргументов и возвращаемых значений операции используют коллекции коллекций. Авторы утверждают, что понятие контекста позволит оптимизировать процесс вычисления запроса. К сожалению, в статье не рассматриваются причины, по которым использованы операции, не ориентированные на работу с XQuery. Перевод запроса на язык алгебры сильно затруднен.

В статье [8] описывается алгебра XAL. Документ в этой алгебре представлен в виде ориентированного графа с корнем с заданными частичными отношениями на ребрах. Операции алгебры работают с множеством узлов и возвращают множества узлов. Главные операции алгебры — selection, projection, product и join.

Главными структурами данных в алгебре Xtasy [9] являются таблицы, похожие на реляционные аналоги. Кортежи в таблице состоят из пар переменных и значений. Таблица строится с помощью операции path, аргументом которой является *input filter* — дерево, описывающее путь в документе, переменные, которые надо присвоить, и способ, по которому будут объединяться результаты из разных мест XML-документа. Обратная операция return, используя в качестве параметра output filter (конструктор элементов и атрибутов), создает XML-документ. Остальные операции — это переработанные операции реляционной алгебры (Join, Selection, Projection и т.д.). Все операции описаны неформально, и лишь простые запросы могут быть переведены на язык алгебры.

В статье [15] описана достаточно простая алгебра, в которой формально определены система типов и синтаксис выражений. Авторы утверждают, что алгебра охватывает семантику многих языков запросов. В качестве

примера приводится перевод выражений XQuery на язык алгебры. К сожалению, алгебра имеет множество недостатков, на которые указывают сами авторы.

Другая алгебра, называемая XAT, описана в статье [16]. Данные в XAT представляются в виде таблиц. Операции алгебры поделены на три группы: XML-операции, SQL-операции и специальные операции. Операции первой группы выполняют действия, привычные для модели данных XML (Navigate, Agregate, Composter и т.д.), операции второй группы — модифицированные SQL-операции (Project, Select, Join, GroupBy и т.д.), операции третей группы предназначены для выполнения специальных действий, таких как итерация по коллекции или выбор. В алгебре отсутствуют операции, создающие или изменяющие существующие элементы. Все операции описаны неформально, с помощью примеров, также неформально описано преобразование выражения XQuery в выражение алгебры. Многие детали при этом упущены.

Авторы статьи [17] основной акцент сделали на четкое формальное описание модели данных и операций. Стандартный запрос XQuery переводится в выражения алгебры достаточно прозрачным образом. В работе рассмотрены основные отличия XML-операций от аналогичных операций реляционных алгебр. Авторы указали пути возможной оптимизации полученных выражений, но сами способы оптимизации в статье не рассматриваются.

4. АЛГЕБРА А.В. ЗАМУЛИНА С РАСШИРЕНИЯМИ ДЛЯ ПРОСТРАНСТВ ИМЕН

Описываемая в статье “An XML algebra for XQuery” [17] алгебра охватывает практически все выражения XQuery. Алгебра использует достаточно простую модель данных, представленную W3C [13]. Это дало возможность определить набор простых операций (нецелесообразно при такой простой модели определять операции высокого порядка, которые используются в других алгебрах), что позволяет упростить перевод запроса XQuery на язык алгебры.

Представленная модель данных описана посредством многоосновной алгебры, что дало авторам возможность формально определить все операции предложенной алгебры. Система типов модели данных включает в себя набор атомарных типов (xs:Boolean, xs:Integer и т.д.) и тип xdt:untypedAtomic, значения которого не определены как значения более

конкретного типа. Элементами коллекций и множеств могут быть только атомарные значения и узлы.

Предполагается, что для каждого атомарного типа существуют операции приведения значений этого типа к значениям других атомарных типов. Система типов также включает в себя конструкторы множеств, последовательностей и объединений $Set(T)$, $Seq(t)$, $Union(t_1,\ldots,t_n)$, где t, $t_1,\ldots,t_n$ — типы, и конструктор перечисления $Enumeration(I_1,\ldots I_n)$, где $I_1,\ldots I_n$ — идентификаторы.

В дополнение к стандартным типам авторы используют тип *запись*. Для типа запись **rec** $p_1:t_1, \ldots, p_n:t_n$ **end** заданы операция конструктора (создает новую запись, используя в качестве параметров значения $v_1,..,v_n$) и операция проекции, возвращающая значение поля записи.

Следующие операции определены для каждого множества: "$\cup$" (объединение), "$\cap$" (пересечение), "$\subset$" (включение), "$\in$" (принадлежность) и count (количество элементов). Если s — последовательность, то $asSet(s)$ — множество, содержащее элементы s без дубликатов. Для одноэлементных последовательностей определена операция $itemize : Seq(t) \to t$, возвращающая элемент последовательности. Количество элементов последовательности обозначается $|s|$. К последовательностям и множествам, содержащим числовые значения применимы операции avg, sum, max и min.

Схема базы данных определяет сигнатуру $\Sigma = (T, F)$. В дополнение к вышеперечисленным операциям F содержит в себе аксессоры узлов, определенные в [13], все имена функций и операций, определенных в [19], имена навигационных операций и некоторые специальные константы. Две следующие функции также являются частью F:

$$document_order : Seq(Node) \to Seq(Node) \text{ и}$$
$$reserve_order : Seq(Node) \to Seq(Node).$$

Эти функции упорядочивают узлы в документном порядке и в порядке, обратном документному, соответственно. Для поддержки понятия пространства имен расширим сигнатуру Σ типом

$$NCBind = \textbf{rec}\ ncn : NCName,\ Uri : anyURI\ \textbf{end},$$

где ncn — префикс пространства имен, Uri — соответствующий префиксу URI, и двумя константами — $NCBindings : Seq(NCBind)$ и $DefaultNC : NCBind$. Эти константы определяют пространство имен по умолчанию и пространства имен, URI для которых задаются статически на этапе трансляции XQuery выражения на язык алгебры.

Используя операции из F можно строить выражения, которые в дальнейшем могут быть вычислены или интерпретированы. Выражения алгебры можно разделить на три группы:

- обычные алгебраические выражения, вычислимые в той же сигнатуре и алгебре, в которой заданы;
- выражения, написанные в одной сигнатуре, но интерпретированные в алгебре обогащенной сигнатуры (например, операция соединения двух реляционных таблиц);
- выражения, чьи интерпретации являются продуктами новой алгебры (например, конструкторы новых узлов).

Интерпретация выражения e в алгебре A обозначена как $[e]^A$, а результат интерпретации — как $\langle A',f \rangle$, где A' — новая алгебра, а f — результат вычисления.

Для каждого узла определены навигационные функции в дополнение к аксессорам. Все эти функции в качестве аргумента используют узел и возвращают последовательность узлов. Примерами навигационных функций могут служить функции нахождения всех предков, потомков, детей и т.д. данного узла. Определение навигационных функций можно найти в статье [17].

Предполагается, что сигнатура алгебры содержит перечислимый тип orderingMode = Enumeration{ordered, unordered} и константу order_mode типа orderingMode. Значение константы указывает на упорядоченность полученной в результате вычислений последовательности.

Перевод FLWOR выражения на язык алгебры выполняется по шагам. Для начала рассмотрим вспомогательные выражения, которые используются для поддержки некоторых выражений XQuery.

Частью выражений XQuery могут быть путевые выражения, которые позволяют задавать позицию в дереве. Эти выражения интерпретируются в алгебре с использованием навигационных функций. Пусть x, y — переменные, s_1 — последовательность узлов, t — атомарный тип или тип узла, s_2 — последовательность элементов типа t, тогда **path**$(y : s_1) / s_2$ — последовательность элементов типа t. Выражение s_1 и s_2 называются *левым шагом* и *правым шагом* соответственно. Левый шаг используется для выбора последовательности узлов, над которой будут произведены вычисления выражения правого шага. Если тип s_2 — атомарный, то результат вычисления путевого выражения будет конкатенацией результатов вычисления s_2 для каждого $y \in s_1$, если же тип s_2 — узел, то мы получим последовательность,

состоящую из результатов вычисления s_2 для каждого y и отсортированную в документном порядке.

Еще одна особенность выражений XQuery — использование кванторов универсальности и существования. Семантики соответствующих выражений алгебры определены стандартно. Используемый синтаксис: **forall**($x_1 : s_1, \ldots, x_n : s_n$)!b и **exists**($x_1 : s_1, \ldots, x_n : s_n$)!b, где $s_1, \ldots, s_n$ — такие последовательности, что каждая следующая может зависеть от предыдущих, b — булевское выражение.

XQuery содержит в себе набор выражений для создания последовательностей и управления ими. Поэтому в алгебре определены следующие операции для работы с последовательностями: **seq**($e_1, \ldots, e_n$) (где $e_1, \ldots, e_n$ — выражения), **range**(e_1, e_2) (в данном случае e_1 и e_2 возвращают целые числа, результат операции — последовательность чисел от e_1 до e_2), **except**(s_1,s_2), union(s_1,s_2), **intersect**(s_1,s_2). Если e_1 и e_2 — выражения типа Seq(anyAtomicType) и Θ — один из символов отношения “=”, “!=”, “<”, “>”, “<=”, “>=”, то $s_1 \Theta s_2$ — выражение типа Boolean (с интерпретацией этих выражений можно ознакомиться в работе [17]).

Алгебра содержит в себе выражения, с помощью которых можно естественным образом записать выражение FLWOR. Рассмотрим их по порядку.

Для поддержки различных видов операций FOR и LET в статье определены три вспомогательных выражения.

1. $<\cdot y : s\cdot>$ — преобразует последовательность s в последовательность записей $s' = (\text{rec } (v) \mid v \in s)$, при этом y принимает значения из s'. Это выражение необходимо для поддержки операции FOR, содержащего в себе одну присваиваемую переменную.
2. $<\cdot y, i : s\cdot>$ — преобразует последовательность s в последовательность записей $s' = \text{rec}(i,s[i])$, где $i=1,\ldots,|s|$. Это выражение необходимо для поддержки операции FOR, содержащей в себе одну присваиваемую переменную и одну позиционную переменную.
3. $<\cdot y = e\cdot>$ — возвращает последовательность, состоящую из записи типа **rec** y: t **end**, где t — тип выражения e. Это выражение необходимо для поддержки операции LET.

Следующее выражение поддерживает любую комбинацию операций FOR и LET: $s_1{*}s_2$, где s_1 — выражение, возвращающее последовательность записей типа **rec** $x_{11} : t_{11}, \ldots, x_{1m} : t_{1m}$ **end**, а s_2 — выражение, возвращающее последовательность записей типа **rec** $x_{21} : t_{21}, \ldots, x_{2n} : t_{2n}$ **end**.

Результат вычисления s_1*s_2 имеет тип Seq(**rec** x_{11} : t_{11}, ..., x_{1m} : t_{1m}, x_{21} : t_{21}, ..., x_{2n} : t_{2n} **end**). Выражение s_1*s_2 вычисляется следующим образом: для каждого элемента из s_1 вычисляется выражение s_2, после чего результаты вычисления s_1 и s_2 заносятся в одну запись (сначала элементы из s_1, потом из s_2), которая добавляется к результирующей последовательности.

Пример: выражение <·x : (1, 2, 3)·>*<·y = (x+1, x+2)·> после вычисления примет следующий вид — (⟨1, (2,3)⟩, ⟨2, (3,4)⟩, ⟨3, (4,5)⟩). Если books — это последовательность книг, то выражение <·x:books·>*<·y:x.child::element(authors)·> будет последовательностью пар, где каждая книга будет в паре с каждым из авторов этой книги. Как мы видим, часть выражения XQuery, содержащая операции FOR и LET, естественным образом интерпретируется в алгебре.

Выражение **selection** используется для выбора части последовательности на основе некоего критерия (операция WHERE). В отличие от реляционных алгебр, такое выражение в XML-алгебре включает в себя проверку узла (node test) в дополнение к проверке предикатом. Проверка узла, в свою очередь, может быть проверкой типа (kind test) или проверкой имени (name test). Проверка узла используется для предварительной выборки узлов из последовательности на основе их имени или типа. Проверка предикатом применяется только к тем узлам, что прошли проверку узла. Каждое выражение **selection** имеет следующую форму: s :: p, где s — последовательность, p — условие выбора. Такие выражения сохраняют порядок в последовательности и не изменяют алгебру.

Пусть s — последовательность узлов, тогда

1) s :: node() — это последовательность, идентичная s,
2) s :: element() — последовательность всех элементов из s,
3) s :: attribute — последовательность атрибутов из s и т.д.,
4) s :: element(n,t) — последовательность элементов типа t с именем n (более подробно с видами kind test можно познакомится, прочитав статью [17]).

Определим выражения проверки имени для последовательности s. Пусть последовательность

$$p : Seq(NCName) = (ncn \mid ncn = DefaultNC.ncn \vee \exists\, nb \in NCBindings,\ ncn = nb.ncn\ )$$

содержит в себе все префиксы известных пространств имен.

1. Если $ncn \in p$ — имя пространства имен, тогда s::element (ncn, *) — последовательность узлов s', определяемая следующим образом:

$s' = (nd \mid nd \in s$ & dm:node-kind(nd) = element &
fn : prefix-from-QName(dm:node-name(nd)) = ncn).

2. Если ln : NCName — локальное имя, тогда s::element (*, ln) — последовательность узлов s', определяемая следующим образом:
$s' = (nd \mid nd \in s$ & dm:node-kind(nd) = element &
fn:local-name-from-QName(dm:node-name(nd)) = ln).
3. Если $ncn \in p$ — имя пространства имен, тогда s::attribute (ncn, *) — последовательность узлов s', определяемая следующим образом:
$s' = (nd \mid nd \in s$ & dm:node-kind(nd) = attribute &
fn:prefix-from-QName(dm:node-name(nd)) = ncn).
4. Если ln : NCName — локальное имя, тогда s::attribute (*, ln) — последовательность узлов s', определяемая следующим образом:
$s' = (nd \mid nd \in s$ & dm:node-kind(nd) = attribute &
fn:local-name-from-QName(dm:node-name(nd)) = ln).

Выражение тестирования предикатом выглядит следующим образом: **select**(y : s) :: p, где y — переменная, принимающая значения из последовательности s, а p — предикат (выражение типа Boolean), зависящий от y. Тогда, если p_i — результат вычисления p для значения $y = v_i$, $[\mathbf{select}(y : s) :: p]^A = \langle A', (v_i \mid p_i)\rangle$. Например, выражение

select(x: books) :: typed-value(x.attribute :: attribute(year)) = 2000

определяет последовательность узлов, содержащих книги, опубликованные в 2000 году.

Операция ORDER BY упорядочивает последовательность записей, полученных в результате выполнения предыдущих операций, основываясь на значениях, вычисленных для каждой записи. В представленной алгебре упорядочивающее выражение сортирует записи на основе одного или нескольких ключей порядка, которые являются пустыми или одноэлементными последовательностями. Синтаксис операций выглядит следующим образом: **stable_order**(e_1,[a_1,b_1,c_1], ..., e_m[a_m,b_m,c_m] : s) и **order**(e_1,[a_1,b_1,c_1], ..., e_m[a_m,b_m,c_m] : s), где s — последовательность, e_1, ..., e_m — ключи сортировки, a_k и b_k — один из символов “↑” или “↓” (a_k указывает на восходящий или нисходящий порядок, b_k — на положение пустых последовательностей) и c_k — пустая строка, если t'_k не является типом string, и, возможно, не пустая иначе. В результате получаем последовательность, содержащую те же элементы, что и исходная, но упорядоченную так, как это задают a, b и c. Второе выражение отличается от первого

только сохранением относительного положения элементов с равными значениями ключей порядка.

Операция RETURN конструкции FLWOR представлена в алгебре в виде выражения *mapping*. Если s — выражение типа Seq(**rec** $x_1 : t_1, ..., x_n : t_n$ **end**) и e — типа Seq(t), то s ► e — выражение типа Seq(t), называемое выражением *mapping*. Для каждого элемента последовательности s вычисляется выражение e и результат добавляется к возвращаемой последовательности. Порядок возвращаемой последовательности совпадает с порядком исходной.

Таким образом, мы можем увидеть, что конструкция FLWOR преобразуется в выражение алгебры достаточно просто, при этом тип записи используется только внутри этого выражения.

Отдельно в статье рассмотрены конструкторы узлов — набор выражений, копирующих узлы или создающих новые узлы. Интерпретация этих выражений меняет алгебру и создает элемент новой алгебры. Поэтому для результата выражения используется нотация ⟨A, v⟩, где A — алгебра, v ∈ |A| — элемент алгебры. Множество всех пар ⟨A, v⟩, где v имеет значение типа t обозначено $A_t(\Sigma)$. Функции fst и snd, применяемые к таким парам, возвращают первый и второй компоненты, соответственно. Предполагается, что сигнатура алгебры содержит перечислимый тип

constructionMode = Enumeration{strip, preserve}

и константу con_mode типа constructionMode, которая управляет значениями некоторых асессоров узлов. Добавим также два перечислимых типа

copynamespacesMode1 = Enumeration{preserve, no-preserve} и
copynamespacesMode2 = Enumeration{inherit, no-inherit}

и константу copyns_mode типа

rec mode1:copynamespacesMode1, mode2: copynamespacesMode2 **end**,

которая определяет, каким образом будут копироваться связи имени пространства имен и самого пространства имен при копировании узлов.

В алгебре определены следующие конструкторы.

1. **copy_node**(s, end). Аргументы: s — выражение типа Seq(Node), end — узел элемента. Результат: копия узла из s со всеми дочерними узлами, который присоединяется к узлу end в качестве дочки.
2. **copy_nodes**(s, end). Аргументы: s — выражение типа Seq(Node), end — узел элемента. Результат: копии узлов из s со

всеми дочерними узлами, которые присоединяется к узлу end в качесте дочки.

3. **attribute_node**(n,e). Аргументы: n — QName, e — String. Результат: новый узел атрибута с именем n и значением e.
4. **text_node**(e). Аргумент: e — String. Результат: тектовый узел со значением e.
5. **document_node**(elseq). Аргументы: elseq — последовательность элементов и Text узлов. Результат: документный элемент с дочерними элементами и текстовыми узлами из elseq.
6. **element_node**(n,atseq,e). Аргументы: n — QName, e — String, atseq — последовательность атрибутов. Результат: узел элемента с атрибутами из atseq, именем n и значением e.
7. **element_node**(n,atseq,elseq). Аргументы: n — QName, elseq — последовательность элементов и Text узлов, atseq — последовательность атрибутов. Результат: узел элемента с атрибутами из atseq, именем n и дочерними элементами и текстовыми узлами из elseq.

Определим конструктор элемента **element_node**(n, atseq, nsseq, elseq). Аргументы: n — QName, elseq — последовательность элементов и Text узлов, atseq — последовательность атрибутов, nsseq — последовательность типа NCBind. Результат: узел элемента с атрибутами из atseq, именем n и дочерними элементами и текстовыми узлами из elseq. Аргумент nsseq используется для задания значения аксессора namespace-bindings, как это показано ниже.

При создании нового элемента (последние три конструктора) значение аксессора namespace-bindings(nd) = N конструируется следующим образом:

1) в N включаются все связи префиксов с пространствами имен из nsseq (только для последнего конструктора);
2) для каждого внешнего конструктора, содержащего последовательность nsseq, в N включаются все связи, которые не переопределены в этом или промежуточных конструкторах;
3) в N добавляется связь для префикса xml с URI http://www.w3.org/XML/1998/namespace;
4) в N включаются все связи для префиксов, используемых в именах атрибутов, содержащихся в atseq, и для имени n, если эти связи не

были добавлены раннее (эти связи определяются с помощью константы NCBindings).

При использовании понятия пространства имен изменяется функция копирования элемента (одна из функций, необходимых для реализации более общей функции **copy_node**). Рассмотрим эту функцию подробнее:

$\mathbf{copy_element_node}^{A}(nd, end) = \langle A', nd' \rangle$,

где nd' — узел, не принадлежащий |A|, A' — расширение алгебры A, построенное следующим образом.

1. Пусть A^1 — расширение алгебры A, такое, что
   - $nd' \in A^1_{Element}$,
     - parent(nd') = end и
       $children^{A1}(end) = children^{A}(end) + (nd')$, если $end \neq NULL$
     - parent(nd') = (), иначе;
   - children(nd') = (), attributes(nd') = (),
   - если con_mode = strip:
     type-name(nd') = xdt:untyped,
     string-value(nd') = string-value(nd),
     typed-value(nd') = string-value(nd'),
     nilled(nd') = false;
   - если copyns_mode.mode1 = no-preserve:
     namespace-bindings(nd') = N, где N содержит в себе все связи имен пространств имен с пространствами имен для всех пространств имен, которые используются в имени копируемого элемента и именах его атрибутов,
   - acc(nd') = acc(nd) для любого другого аксессора acc.
2. $A^{at} = fst(\mathbf{copy_nodes}^{A1}(attributes(nd), nd'))$;
   тогда $A' = fst(\mathbf{copy_nodes}^{Aat}(children(nd), nd'))$.
   Формальные определения остальных конструкторов можно найти в статье.

Представленная в статье алгебра является набором простых выражений, с помощью которых строятся более сложные операции. Этот набор существенно отличается от операций реляционных алгебр, в основном из-за более сложной структуры XML-документа. Только одна операция — select — немного похожа на описываемую в статье, но только той частью, где происходит проверка предиката. Операция project заменена путевыми выражениями, операция join — набором выражений, позволяющем создавать поток записей на основе нескольких, возможно, вложенных частей доку-

мента. Также определены выражения, позволяющие создавать новые узлы. Побочным эффектом этих выражений является создание новой алгебры, и так как в XQuery возможны вложенные выражения, то любое выражение может обладать таким побочным эффектом. Еще одной особенностью алгебры является задание контекста первой операцией в сложном выражении для второй операции, что может использоваться при оптимизации вычислений запроса.

СПИСОК ЛИТЕРАТУРЫ

1. XQuery 1.0: An XML Query Language, W3C Candidate Recommendation, 3 November 2005. — http://www.w3.org/TR/xquery/.
2. Jagodish H. V., Al-Khalifa Sh., Chapman A., et al. TIMBER: A Native XML Database // The VLDB J. — 2002. — Vol. 11, N 4. — P. 274–291.
3. Jagodish H. V., Lakshmanan L. V. S., Srivasta D., Thompson K. Tax: A Tree Algebra for XML // Proc. Internat. Workshop on Databases and Programming Languages, Marino, Italy, Sept., 2001. — P. 149–164.
4. Paparizos S., Al-Khalifa Sh., Jagodish H. V., et al. A Physicial Algebra for XML. — http://www-personal.umich.edu/spapariz/publications.html.
5. Chen Zh., Jagodish H. V., Lakshmanan L. V.S., Paparizos S. From Tree Patterns to Generalized Tree Patterns: On Efficient Evaluation of Xquery // Proc. VLDB Conf., Berlin, Germany, Sept., 2003.
6. Paparizos S., Wu Yu., Lakshmanan L. V. S., Jagodish H. V. Tree Logical Classes for Efficient Evaluation of XQuery // Proc. SIGMOD Conf., Paris, France, Jun., 2004.
7. Viglas S. D., Galanis L., D. DeWitt J., et al. Putting XML Query Algebras into Context. — http://www.cs.wisc.edu/niagara.
8. Frasincar F., Houben G.-J., Pau C. Xal: an algebra for xml query optimization // Database Technologies. — 2002.
9. Sartiani C., Albano A. Yet Another Query Algebra For XML Data // IDEAS. — 2002. — P. 106-115.
10. Kalchenko V. Review of algebras for XML database systems. — Novosibirsk, 2005. — (Prepr. / IIS SB RAS; N 127).
11. Duerst M., Suignard M. Internationalized Resource Identifiers (IRIs), January 2005. — http://www.rfc-editor.org/rfc/rfc3987.txt.
12. Ehrig H., Mahr B. Fundamentals ofAlgebraic Specifications 1, Equations and Initial Semantics // EATCS Monographs on Theoretical Computer Science. — 1985. — Vol. 6.
13. XQuery 1.0 and XPath 2.0 Data Model (XDM), W3C Candidate Recommendation, 3 November 2005. — http://www.w3.org/TR/xpath-datamodel/.
14. Namespaces in XML 1.0 (Second Edition), W3C Recommendation, 16 August 2006. — http://www.w3.org/TR/xml-names/.

15. Fernandez M., Simeon J., Wadler Ph. An Algebra for XML Query // Proc. FST TCS, Delhi, Dec., 2000.
16. Zhang X., Rundensteiner E. XAT: XML Algebra for Rainbow System. — Worcester, 2002. — (Tech. Rep. / Worcester Polytechnic Institute; WPI-CS-TR-02-24).
17. Novak L., Zamulin A. An XML-algebra for XQuery. — Novosibirsk, 2005 — (Prepr. / IIS SB RAS; N 125).
18. XML Path Language (XPath) 2.0, W3C Candidate Recommendation, 3 November 2005. — http://www.w3.org/TR/xpath20/.
19. XQuery 1.0 and XPath 2.0 Functions and Operators, W3C Candidate Recommendation, 3 November 2005. — http://www.w3.org/TR/xpath-functions/.
20. Extensible Markup Language (XML) 1.0 (Third Edition), W3C Recommendation, 04 February 2004. — http://www.w3.org/TR/2004/REC-xml-20040204/.
21. XML Schema Part 0: Primer Second Edition, W3C Recommendation, 28 October 2004. — http://www.w3.org/TR/xmlschema-0/.

А.Б. Пятков

ФОРМАЛЬНАЯ МОДЕЛЬ ОСНОВНЫХ ПОНЯТИЙ ЯЗЫКА C#[1]

1. ВВЕДЕНИЕ

Характерным свойством работ [1, 2, 3, 4], посвященных формализации семантики объектно-ориентированных языков программирования, является их большая сложность. Например, в работе Боргера и др. [1] средствами ASM описан не сам язык C# [6], а его абстрактный интерпретатор, вследствие чего работа получилась сложной для понимания. А в работе Джакобса и Пола [2] используются моноиды и коалгебры для описания семантики выражений и операторов языка, что затрудняет понимание ввиду сложности математического аппарата. Чтобы избежать недостатков этих работ, мы будем использовать лишь простейшие понятия теории множеств и многоосновных алгебр, что упрощает описание и делает его понятным не только специалистам-математикам, но и программистам. Подобный подход используется в работе [5], но там он используется для описания абстрактного императивного языка программирования.

В силу ограниченности объема статьи не рассматривается параллелизм, обработка исключений, атрибуты, вложенные типы, пространства имен и некоторые другие особенности языка. Кроме того, описание алгебры программы (разд. 3) дано схематично.

2. ОБЪЕКТНАЯ МОДЕЛЬ ДАННЫХ

2.1. Система типов

Наша модель основывается на следующей системе типов, которые могут встретиться в любой программе на языке C#:

T ::= BASE | ENUM | INTERFACE | CLASS | STRUCT | DELEGATE | ARRAY,

[1] Работа выполнена при поддержке РФФИ, грант 04-01-00272.

где BASE, ENUM, INTERFACE, CLASS, STRUCT, DELEGATE — непересекающиеся множества соответственно имен базисных типов, перечислимых типов, имен интерфейсов, имен классов, структур и делегатов, а ARRAY — конструктор имен типов массивов. Они делятся на 2 группы — *типы-значения* (BASE, ENUM, STRUCT) и *типы-ссылки* (CLASS, INTERFACE, DELEGATE, ARRAY).

Везде ниже T* — множество всех последовательностей элементов множества T, включая пустую последовательность <>, а T^{v} — множество $T \cup \{void\}$, где void — специальный тип, не принадлежащий T, 2^{CLASS} — множество подмножеств типа CLASS. Для обозначения произвольного имени интерфейса мы будем использовать букву i, для класса — c, для структуры — s, а для имени интерфейса/класса/структуры — сочетание i/c/s.

2.2. Иллюстративный пример

Для пояснения вводимых понятий мы будем использовать набор объявлений интерфейсов, классов, структур, делегатов (рис.1). Ключевое слово **this** — это ссылка на текущий объект класса, а ключевое слово **value** — это параметр, передаваемый в функцию доступа. Все остальные ключевые слова либо объясняются ниже (например **virtual**, **override**), либо являются стандартными для многих языков программирования (например **return** или **new**).

2.3. Схема классов

Множество объявлений классов, структур и интерфейсов мы называем схемой классов. Это понятие в некоторой степени эквивалентно понятию сигнатуры, или словаря в обычных алгебрах. Для формального определения схемы классов введем сначала понятие собственного объявления класса.

Класс состоит из описаний *констант* (constant), *полей* (field), *методов* (method), *свойств* (property), *событий* (event), *индексаторов* (indexer), *операций* (operator), *конструкторов и деструктора*. Константы и поля описывают данные класса, а все остальные члены — его поведение. Кроме того, класс может содержать вложенные типы (в том числе вложенные классы), которые в нашей модели не рассматриваются.

Методы объявляются с указанием имен, типов параметров и типа результата. Конструкторы объектов (специальные методы, используемые для создания и инициализации объекта) обозначаются именем класса и не имеют результата, статический конструктор не имеет ни типа, ни результата и

используется для инициализации статических данных класса. Свойства объявляются с указанием имени, типа и функций доступа get и/или set. Индексеры объявляются с указанием типа параметра и функций доступа. Деструктор может быть только один, не имеет параметров и вызывается автоматически при уничтожении объекта. Имена, следующие за ":" в объявлении интерфейса, указывают на базовые интерфейсы, а в объявлении класса — либо на базовые интерфейсы, либо на базовый класс (множественное наследование классов в C# недопустимо).

```
interface IPaintable { void Paint(); }
interface IFigure : IPaintable { Point Location { get; set; } }
struct Point {
    Point(int x, int y) { X = x; Y = y; }
    public int X, Y;
}
abstract class Figure2D : IFigure {
    public abstract double Perimeter { get; }
    public abstract Point Location { get; set;}
    public virtual void Paint(){};
}
class Circle : Figure2D {
    public override double Perimeter { get { return 0; } }
    public double Radius = 0;
    public override void Paint() { }
    protected Point m_pCentre = new Point();
    public override Point Location {
        get { return m_pCentre; }
        set { m_pCentre = value; }
    }
}
sealed class Rectangle : Figure2D {
    public Rectangle(int width, int height, point corner) { };
    public override double Perimeter { get { return 0; } }
    protected Point m_pTopCorner = new Point();
    public int Width, Height;
    public new void Paint() { }
    public override Point Location {
        get { return m_pTopCorner; }
        set { m_pTopCorner = value; }
    }
}
```

```
public delegate void AddEventHandler(object sender,
    System.EventArgs e);
public delegate void RemoveEventHandler(object s,
    System.EventArgs e);
class FigureCollection {
    public FigureCollection() { }
    IFigure[] m_array;
    public event AddEventHandler ItemAdded;
    public event RemoveEventHandler ItemRemoved;
    public void Add(IFigure f) {ItemAdded(this,
        new System.EventArgs());}
    public void Remove(IFigure f){ItemRemoved(this,
        new System.EventArgs());}
    public IFigure this[int index] {
        get { return m_array[index]; }
        protected set { }
    }
}
class Picture
{
    private static Picture m_Inst = null;
    static Picture Instance { get { return m_Inst; } }
    static Picture() {
        m_Inst = new Picture();
        m_Inst.collection = new FigureCollection();
        m_Inst.collection.ItemAdded +=
            new AddEventHandler(MItemAdded);
        m_Inst.collection.ItemRemoved +=
            new RemoveEventHandler(MItemAdded);
    }
    static void MItemAdded(object sender, System.EventArgs e)
        { Instance.Refresh(); }
    protected FigureCollection collection;
    internal void Refresh() { }
}
```

Рис. 1. Иллюстративный пример

Множество имен членов класса, требующих хранения в памяти, обозначим VARIABLE. Это множество содержит в себе переменные, константы, делегаты и события. Множество имен всех остальных членов класса обозначим FUNCTION. Пусть $c \in$ CLASS является именем класса, тогда собственное объявление класса — это набор следующих частичных функций:

- field(c) : VARIABLE $\to$ T — объявление полей,
- constant(c) : VARIABLE $\to$ T — объявление констант,
- method(c) : FUNCTION $\times$ $T^* \to 2^{CLASS} \times T^v$ — объявление методов,
- constructor(c) : $T^* \to 2^{CLASS}$ — объявление конструкторов,
- property(c) : FUNCTION, T
  get: FUNCTION $\to 2^{CLASS} \times T$
  set: FUNCTION $\times T \to 2^{CLASS}$,
- event(c) : VARIABLE, DELEGATE
  add : FUNCTION $\to 2^{CLASS}$
  remove : FUNCTION $\to 2^{CLASS}$,
- indexer(c) : FUNCTION, T*, T
  get : FUNCTION $\times (T^* \setminus <>) \to 2^{CLASS} \times T$
  set : FUNCTION $\times (T^* \setminus <>) * T \to 2^{CLASS}$,
- operator(c) : FUNCTION $\times T^v \to 2^{CLASS} \times T$,
- destructor(c) : FUNCTION $\to <>$.

Необходимо отметить, что функции доступа в C# не имеют имен, поэтому мы вводим их самостоятельно. Так как интерфейс не содержит полей, констант, конструкторов и деструкторов, то field(i) = Ø, constant(i) = Ø, constructor(i) = Ø, destructor(i) = Ø. Структуры, в свою очередь, не содержат деструктора, таким образом, destructor(s) = Ø.

Например, собственное объявление класса Rectangle выглядит так:

field (Rectangle) = {(m_pTopCorner → Point), (Width → int), (Height → int)}
constructor (Rectangle) = {(int, int, Point → 0)}
method (Rectangle) = {(Paint, (void) →(0, void))}
property (Rectangle) = {(Perimeter, double : get;), (Location, Point : get; set;)}

Таким образом, каждое объявление переменной или константы вводит ее имя x и тип t. Каждое объявление конструктора вводит строку типов параметров r. Каждое свойство/индексер вводит его имя p/i, тип t и функции доступа get и/или set. Деструктор может быть только один, не имеет параметров и вызывается автоматически. Каждое объявление операции вводит знак операции (стандартная операция языка, которая может быть переопределена), типы аргументов и тип результата t. Каждое объявление метода вводит имя метода m, строку типов параметров r и тип результата. Каждое объявление свойства вводит его имя p и тип т и одну или две функции доступа, объявление индексера вводит имя i, строку параметров r, тип t

и одну или две функции доступа, операция — знак стандартной операции языка, тип параметра t и тип результата, а событие — имя e, тип d и, возможно, две функции для переписывания добавления/удаления обработчиков этого события. Пара (r, t) называется профилем метода, а пара (m, r) — сигнатурой метода. В дальнейшем будем обозначать элементы отображений field, constant парами (x, t), элементы отображения constructor — (r), элементы отображения method — тройкой (m, r, t). Остальные члены класса обозначаются аналогичным образом.

Пусть CDECL обозначает множество собственных объявлений класса, SDECL — множество собственных объявлений структуры, а IDECL — множество собственных объявлений интерфейса. Тогда схема классов Sch состоит из:

- конечного подмножества D множества DELEGATE,
- конечного подмножества I множества INTERFACE,
- конечного подмножества C множества CLASS, включающего имя object,
- конечного подмножества S множества STRUCT,
- конечного множества AR $\in$ ARRAY(T) ,
- бинарного ацикличного отношения isaI на I,
- бинарного отношения isaC на C, представляющего собой дерево с корнем object,
- бинарного отношения impl на $C \cup S \times I$,
- трех тотальных функций cdecl: $C \to$ CDECL, $S \to$ SDECL и $I \to$ IDECL,

так что:

1) любой тип является одним из вышеперечисленных;

2) имена констант, полей, свойств и событий должны быть уникальными;

3) в классе может быть объявлена либо переменная, либо константа, либо свойство с данным именем;

4) имя метода должно отличаться от имен всех не методов класса. Кроме того, все сигнатуры методов должны быть одинаковы (модификаторы параметров ref и out не являются различием);

5) сигнатуры индексеров и операторов должны быть различны.

Для нашего примера указанные множества и отношения выглядят следующим образом:

```
I = { Ipaintable, IFigure }
C = { Point, Figure2D, Circle, Rectangle, FigureCollection, Picture }
AR = {ARRAY(IFigure)}
isaI = { (IFigure, Ipaintable)} isaC = {  }
isaC = { (Point, object), (Figure2D, object), (Circle, Figure2D),
     (Rectangle, Figure2D), (FigureCollection, object), (Picture, object) }
Impl =  { (Figure2D, IFigure), (Circle, IFigure), (Rectangle, IFigure) }
```

2.4. Делегаты и события

Делегат(delegate) — это класс, унаследованный от стандартного класса System.Delegate. Его назначение такое же, как и у указателя на функцию в других языках (например, C++). Объявление нового типа делегата вводит тип возвращаемого значения, имя делегата и список параметров: Delegate() : T^v, DELEGATE, T*, кроме того он может иметь модификаторы **public** и **internal**.

Множество методов, скрытых в делегате, называется «список вызова» (invocation list). При создании экземпляра делегата в качестве параметра ему передается метод класса, объекта либо структуры с сигнатурой, совпадающей с сигнатурой объявления типа делегата. К делегатам применяется две операции — “+” и “–”, которые, фактически, являются операциями сложения и вычитания на его списке вызова. Таким образом, с помощью одного делегата можно вызвать последовательность разнообразных методов, которые будут вызываться в порядке очереди, причем если вызов делегата включает в себя какой-либо ссылочный параметр, то он может меняться в процессе выполнения методов. Еще одной особенностью делегатов являются анонимные методы. При создании экземпляра делегата вместо метода объекта/класса/структуры ему можно передать тело метода в качестве параметра. Например, это может выглядеть следующим образом:

```
delegate bool Action(Node n); // объявление типа делегата Action.
static void Walk(Node n, Action a) { // - объявление метода Walk
while (n != null && a(n)) n = n.Next; }
     // вызов метода Walk, с использованием анонимного метода
Walk(list, delegate(Node n){ Console.WriteLine(n.Name); return true;});
```

Основной особенностью анонимных методов, ради которой они обычно и используются, является то, что эти методы имеют доступ ко всем локальным переменным, чего нет у именованного метода.

Событие (event) — это член класса, который имеет тип делегата. К нему можно добавлять обработчики (см. список вызова), которые вызовутся при его создании.

2.5. Отношения наследования

Нетрудно заметить, что отношение isaI определяет граф множественного наследования на множестве интерфейсов I, а отношение isaC — дерево одиночного наследования на множестве классов C. Отношение impl определяет граф реализаций, в котором имя класса/структуры может быть связано с одним или несколькими именами интерфейсов. Например, интерфейс IFigure наследует IPrintable, а класс Rectangle наследует Figure2D.

Определим:

- isaI(i) = {i′ | i isaI i′} — множество суперинтерфейсов интерфейса i,
- isaC(c) = {c′ | c isaC c′} — суперкласс класса c,
- impl(c) = {i | c impl i} — множество интерфейсов, реализуемых классом c,
- impl(s) = {i | s impl i} — множество интерфейсов, реализуемых структурой s.

Тогда для каждого имени интерфейса i ∈ I кортеж (i, isaI(i), method(i), property(i), event(i), indexer(i), operator(i)) — это полное объявление интерфейса с именем i. Для каждого имени класса *c* ∈ *C* кортеж (c, isaC(c), impl(c), field(c), constant(c), constructor(c), destructor(c), method(c) property(c), event(c), indexer(c), operator(c)) — это полное объявление класса с именем c. Для каждого имени структуры s ∈ S кортеж (s, impl(s), field(s), constant(s), constructor(s), method(s), property(s), event(s), indexer(s), operator(s)) — полное объявление структуры с именем s.

Прямые результаты приведенных определений.

1. Модель поддерживает одиночное наследование классов, множественное наследование интерфейсов и множественные реализации интерфейсов.
2. Не допускается совмещение имен членов в пределах класса или интерфейса, кроме совмещения имен методов при условии их разных сигнатур.

Напоминаем, что определенная выше схема классов является аналогом сигнатуры в обычных алгебрах и потому не содержит тел методов и конструкторов. Снабжение их телами приводит нас к спецификации классов, являющейся аналогом алгебраической спецификации.

2.6. Отношения тип—подтип

Каждая схема естественным образом определяет строгий порядок над классами и интерфейсами, который мы называем отношением тип-подтип и обозначаем $<_{isa}$. Оно определяется рекурсивно следующим образом:

- если c isaC c', тогда $c <_{isa} c'$,
- если i isaI i', тогда $i <_{isa} i'$,
- если c impl i, тогда $c <_{isa} i$,
- если s impl i, тогда $s <_{isa} i$,
- если $ic <_{isa} ic'$ и $ic' <_{isa} ic''$, тогда $ic <_{isa} ic''$,
- если $s <_{isa} i'$ и $i' <_{isa} i''$, тогда $s <_{isa} i''$.

Например,
$Figure2D <_{isa} IFigure$, $IFigure <_{isa} IPaintable$, $Rectangle <_{isa} Figure2D$ и т.д. Если $ic <_{isa} ic'$, тогда ic называется подтипом ic', а ic′ — супертипом ic.

C# предлагает унифицированную систему типов. Все классы наследуются от стандартного класса **object**, т.е. $c <_{isa} object$. Кроме того, вообще любой тип, который может существовать в программе, в том числе и базовый, считается классом object (т. е. может быть преобразован к нему автоматически). Таким образом, становится возможным вызывать методы класса object даже для базовых типов, таких как int.

2.7. Классификация интерфейсов, классов и структур

Если класс объявлен как **abstract**, то он называется абстрактным классом, для которого не может быть создано ни одного объекта. Кроме того, он может содержать абстрактные методы. Структура и интерфейс не могут быть абстрактными по очевидным соображениям. Например, класс $Figure2D \in abstract_class$.

Если класс объявлен как **sealed**, то от этого класса не может быть унаследован никакой другой класс. Структуры и интерфейсы также не могут быть объявлены с этим модификатором. В нашем примере $Rectangle \in sealed_class$.

Если класс объявлен как **static**, то он называется статическим. Это значит, что от него нельзя наследоваться, он унаследован напрямую от класса object, он не может содержать операторов, не может иметь членов с **protected** или **protected internal** доступом и может содержать только статические члены. С этим модификатором нельзя объявлять структуры и интерфейсы.

Модификаторы доступа **public**, **private**, **protected**, **internal**, **protected internal** определяют область видимости класса, структуры либо интерфейса. Модификаторы **private**, **protected** и **protected internal** могут быть объявлены только для вложенных классов/структур/интерфейсов. У любого класса, структуры либо интерфейса всегда есть модификатор доступа (если явно никакой не указан, то используется модификатор по умолчанию).

sealed_class ∩ abstract_class = abstract_class ∩ static_class =
static_class ∩ sealed_class = Ø,

т. е. никакие из модификаторов **abstract**, **sealed**, и **static** не могут быть объявлены одновременно.

2.8. Классификация элементов классов, структур и интерфейсов

Компоненты классов, структур и интерфейсов подразделяются на несколько непересекающихся групп согласно ряду ортогональных классификаций.

По первой классификации элементы класса/структуры делятся на static_members(cs) и instance_members(cs) — члены класса/структуры и члены экземпляра соответственно, т. е.

members(cs) = static_members(cs) ∪ instance_members(cs).

Элементы с модификатором **static** являются членами класса/структуры, все остальные — члены объекта. Константы, индексеры и деструкторы могут быть только элементами объекта, т. е.

static_constant(cs) = static_indexer(cs) = static_destructor(c) = Ø.

Члены интерфейсов на эти категории не делятся, так как модификатор static в интерфейсах не употребляется.

По другой классификации в классе могут существовать четыре непересекающихся подмножества элементов: abstract_members(c), override_members(c), virtual_members(c) и new_members(c), называемых соответственно абстрактными, подменяющими, виртуальными и скрывающими. Эти модификаторы связаны с наследованием, поэтому применимы для методов, свойств, индексеров и событий. Абстрактный элемент может быть объявлен только в абстрактном классе, где он помечается как **abstract**. Для того чтобы можно было создать экземпляр класса, абстрактный элемент обязательно должен быть подменен. Подменяющий элемент класса — это тот, который подменяет абстрактный либо виртуальный элемент. Он может быть помечен ключевым словом **sealed**, это означает, что он будет наследоваться напрямую из этого класса. Если элемент виртуаль-

ный, то это означает, что он может быть подменен. Скрывающий элемент — это тот, который скрывает элемент, объявленный в базовом классе. Скрывающий элемент отмечается модификатором **new**. Абстрактный элемент должен быть подменен, следовательно, его нельзя скрыть. Все методы, свойства индексеры и события в интерфейсах являются абстрактными, а в структурах — не принадлежат ни одной из этих групп.

По третьей классификации элементы класса подразделяются на **public**, **internal**, **protected**, **private** и **protected internal** элементы с различными правами доступа. Таким образом, элементы класса могут быть представлены следующим образом:

$$\text{members}(c) = \text{private_members}(c) \cup \text{internal_members}(c) \cup \text{protected_members}(c) \cup \text{public_members}(c) \cup \text{protected_internal_members}(c),$$

$$\text{abstract_member}(c) \cap \text{private_member}(c) = \varnothing,$$

так как абстрактный метод должен быть подменен.

Кроме того, поля могут иметь модификатор **readonly**, т.е. иметь доступ только на чтение.

2.9. Скрытие, подмена и наследование

Элемент суперкласса может быть скрыт, подменен или наследован. По умолчанию все элементы наследуются, т. е. остаются в точности такими, какими были в суперклассе, в том числе и сохраняя все модификаторы. Скрытие — это создание нового элемента класса с именем уже существующего, при этом старый элемент наследуется, но в новом классе его не видно (зато видно, например, при преобразовании класса к суперклассу). Не нужно путать скрытие с перегрузкой (определение элемента с тем же именем, но другими параметрами), при перегрузке элемент суперкласса будет виден потомку, да и к наследованию перегрузка не имеет никакого отношения. Скрывать можно не только элементы классов, но и элементы интерфейсов. Подмена же — это переписывание тела исходного элемента, так что тот же самый элемент начинает выполнять совершенно другие операции.

Будем говорить, что поле (x, t) класса/интерфейса ic' близко классу c, если $c <_{isa} c'$, (x, t) — неприватное поле в c' и нет такого класса c'' и типа t', что $c <_{isa} c'' <_{isa} c'$ и $(x, t') \in (\text{field}(c'') \cup \text{constant}(c''))$. Точно также метод (m, r, t) класса/интерфейса ic' близок классу/интерфейсу ic, если $ic <_{isa} ic'$, (m, r, t) – неприватный метод в ic' и нет такого класса/интерфейса ic'' типа t' и множества имен классов q', что $ic <_{isa} ic'' <_{isa} ic'$ и

(m, r, t') ∈ method(ic"). Таким образом, поле, близкое некоторому классу, объявлено в каком-то его суперклассе и не переобъявлено ни в каком промежуточном классе; то же самое относится и к методам, только для них распространяется и на интерфейсы. Тогда определения скрытия, подмены и наследования выглядят следующим образом.

- Поле (x, t') класса ic', близкое классу ic, скрыто в ic, если существует (x, t) ∈ (field(ic) ∪ constant(ic)). Класс наследует близкое ему поле (x, t') класса ic', если не существует (x, t) ∈ (field(ic) ∪ constant(ic)).
- Метод (m, r, t′) ∈ cmethod(c′) (т.е. метод класса), близкий классу c, считается скрытым в c, если существует (m, r, t) ∈ cmethod(c). Метод объекта (m, r, t') ∈ cmethod(ic'), близкий классу/интерфейсу ic, считается подмененным (реализованным) в классе/интерфейсе ic, если существует (m, r, t) ∈ imethod(ic).
- Класс/интерфейс ic наследует близкий ему метод (m, r, t') класса/интерфейса ic', если не существует (m, r, t) ∈ method(ic).

Только методы, помеченные ключевыми словами **abstract** и **virtual**, могут быть подменены, причем абстрактный метод должен быть подменен обязательно. Подменяющий метод помечается ключевым словом **override**, кроме того он может быть **sealed** и тогда уже его, в свою очередь, нельзя будет подменить. Скрывающий метод либо переменная помечаются ключевым словом new. При этом можно менять тип результата (для поля — тип переменной) и область видимости (например, изменить **public** на **private**). Абстрактный метод скрыть нельзя, так как он должен быть унаследован.

Все, что описано для методов, верно также и для свойств, индексеров и событий.

Например, метод Paint() класса Rectangle скрывает метод Paint из класса Figure2D и реализует свойство Location.

2.10. Иерархическая корректность

Поскольку метод с именем m может быть объявлен в классе c, в его суперклассе и/или в одном или нескольких интерфейсах, реализуемых классом c или его суперклассом, необходимо убедиться, что нет противоречий между сигнатурами этих методов. Таким образом, приходим к понятию иерархической корректности сигнатуры классов. Следуя спецификации языка мы говорим, что схема классов является иерархически корректной, если:

- если $(m, r, t) \in method(ics)$ и если $(m, r, t') \in method(ics) \Rightarrow t = t'$, т.е. класс, интерфейс или структура не могут содержать два метода с одинаковой сигнатурой, но разным типом результата;
- $c <_{isa} c' \Rightarrow c' \notin sealed_class$, т. е. **sealed** класс не может иметь подклассов;
- если $(m, r, t) \in sealed_method(c)$, $c' <_{isa} c$, то $(m, r, t) \in sealed_method(c')$, т. е. **sealed** метод не может быть подменен в подклассе;
- $(m, r, t) \in sealed_method(c)$, $c <_{isa} c' \Rightarrow$ $(m, r, t) \in virtual_method(c')$ или $(m, r, t) \in abstract_method(c')$, т. е. **sealed** метод можно унаследовать только от виртуального или абстрактного метода.
- если $(m, r, t) \in abstract_method(c)$ и $c' <_{isa} c$, то не существует $(m, r, t) \in new_method(c')$, т. е. абстрактный метод не может быть скрыт;
- если $(m, r, t) \in abstract_method(c) \Rightarrow c \in abstract_class$, т. е. абстрактный метод может быть только в абстрактном классе;
- метод, подменяющий или реализующий метод, не должен быть более доступен, чем базовый, т. е., например, **internal** метод не может быть подменен **public** методом;
- если $(m,r,t) \in virtual_method(c)$ или $(m,r,t) \in abstract_method(c) \Rightarrow$
- $(m,r,t) \in public_method(c) \cup internal_method(c)$, т. е. виртуальный/абстрактный метод должен быть public или internal;
- если $c <isa\ c'$ и $c' \in internal_class \Rightarrow c' \in public_class$, т. е. public класс не может быть унаследован от internal класса;
- структура не может быть унаследована.

2.11. Замыкание схемы

Для любого класса/структуры cs и поля x мы определим частичную функцию ResF(cs, x), вырабатывающую имя класса/структуры/интерфейса, где поле x объявлено или откуда может однозначно быть унаследовано:

- ResF(cs, x) = cs, если существует $(x, t) \in field(cs) \cup constant(cs)$. Это означает, что x объявлен в cs и скрывает x, объявленное в любом суперклассе/суперинтерфейсе, если оно существует. Например, ResF(Width, Rectangle) = Rectangle, потому что оно объявлено именно в этом классе;

- ResF(c, x) = ResF(c', x), если не имеет места предыдущий случай, но существует c', такой что
  (c isaI c' $\vee$ c isaC c' $\vee$ c impl c') & ResF(c', x) определен, и при этом, если ResF(c'', x) определен для некоторого другого c'', удовлетворяющего тем же условиям, то ResF(c', x) = ResF(c'',x). Значит c однозначно наследует x из c';
- ResF(cs, x) не определен, если ни один из предыдущих случаев не имеет места. Это означает, что либо x не объявлен в cs, либо x наследуется из двух или более классов (наследоваться можно только от одного класса)/интерфейсов. Например, не определен ResF(Rectangle, Length).

Таким образом, если ResF(cs, x) = cs', то либо cs' = cs и x объявлено в cs, либо cs $\neq$ cs' и x наследуется из cs'. Поэтому, если (x, t) — объявление переменной в cs', мы можем расширить множество field(cs) полем (x, t), определив $\overline{\text{field}}$(cs) = (x, t), и если (x, t) — объявление константы в cs', мы можем расширить множество constant(cs) константой (x, t), определив $\overline{\text{constant}}$(cs) = (x, t).

Аналогично для класса/интерфейса/структуры ics, имени метода m и строки типов r в иерархически корректной схеме классов мы определим частичную функцию ResM(ics, m, r), вырабатывающую имя класса/интерфейса/структуры, где метод m объявлен или имя класса/интерфейса откуда он может быть однозначно унаследован:

- ResM(ics, m, r) = ics, если существует (m, r, t) $\in$ method(ics). Это означает, что в ics объявлен метод с сигнатурой (m, r) и он скрывает/подменяет соответствующий метод в суперклассах/суперинтерфейсах класса/интерфейса/структуры ic, если таковые есть.
- ResM(ics, m, r) = ic', если не имеет места предыдущий случай, но существуют такие ic' и (m, r, t), что ics $<_{isa}$ ic' и
  (m, r, t) $\in$ method(ic').
  Это означает, что ics однозначно наследует m.
- ResM(ics, m, r) не определен в других случаях.

Таким образом, если (m, r, t) — метод в ic', мы можем расширить множество method(ics) методом (m, r, t), определив таким образом $\overline{\text{method}}$(ics).

Подмножества field(c), constant(c), method(ic) каждого класса c и каждого класса/интерфейса ic расширяются соответствующим образом, чтобы

включить наследуемые компоненты. Полученную схему мы называем замыканием, и легко доказать, что если схема классов иерархически корректна, то иерархически корректно и ее замыкание $\overline{\mathrm{Sch}}$. Только иерархически корректные схемы будут рассматриваться в дальнейшем. Для делегатов всего этого не требуется, так как все их методы наследуются из класса System.Delegate.

Например, все классы расширяются методом класса object — ToString().

3. АЛГЕБРА ПРОГРАММЫ

3.1. Схема программы

Схема программы определяет символы, которые могут использоваться в данной программе. Она формируется на основе типов, описанных в схеме Sch. Никакие другие типы не могут быть использованы. Необходимо отметить, что есть типы, которые описаны, например, в библиотеках классов. Эти типы также входят в нашу схему классов.

3.2. Состояние программы

Состояние программы формально представляется многоосновной алгеброй, компонентами которой являются множества значений типов данных, операции примитивных типов данных, функции доступа, ячейки переменных и т.п. В каждой такой алгебре выделяются *статическая часть* (базисная алгебра), состоящая из значений и операций примитивных типов данных, классов и интерфейсов, и *динамическая часть*, состоящая из функций доступа, объектов и динамически создаваемых переменных.

3.3. Базисная алгебра

Базисная алгебра B связывает с каждым типом t некоторое множество, его носитель, и с каждой операцией типа t — частичную функцию, ее реализацию. Носители наших типов определяются следующим образом:

- носитель базисного типа t — это некоторое множество B_t;

- носитель каждого типа ARRAY(t), каждого класса $c \in C$ и каждой структуры $s \in S$ — специальное множество Ref, элементы которого называются ссылками;
- носитель типа **void** — одноэлементное множество.

Носители всех типов — непересекающиеся множества. Существует также специальное значение **null**, не принадлежащее никакому носителю.

Каждая операция каждого базисного типа $op: t_1, \ldots t_n \to t$ реализуется как функция $op^B: B_{t1} \times \ldots \times B_{tn} \to B_t$, когда $n > 0$ (для языка C# $n \le 3$), и как константа $op^B \in B_t$ в противном случае.

Единственной предопределенной операцией над классом object является операция равенства “==”, которая имеет значение **true**, когда оба объекта имеют один и тот же адрес (т. е. являются одним и тем же объектом). Эту операцию можно применять к любым типам, предварительно преобразовав их к типу **object**.

3.4. Алгебра состояния

Алгебра состояния A программы P схемы Sch определяется следующим образом.

- $A_t = B_t$ для любого базисного типа t и $op^A = op^B$ для каждой операции/константы базисного типа, т. е. алгебра состояния расширяет базисную алгебру.
- Конечное множество $A_{ics} = A^O_{ics} \cup \{null\}$, где $A^O_{ics} \in Ref$, связывается с каждым ics, так что если $ics <_{isa} ic'$, то $A^O_{ic'} \subseteq A^O_{ics}$, а в противном случае объект $o \in A^O_{ic'}$ & $o \in A^O_{ics} \Leftrightarrow$ существует ic'' такой, что $ic'' <_{isa} ic'$ и $ic'' <_{isa} ics$, т. е. в каждом состоянии есть множество ссылок, так что множество ссылок супертипа включает в себя ссылки подтипов и множества не пересекаются, если не связаны отношением тип—подтип и не имеют общих подтипов.
- Частичная функция $x^A_{cst} : A_{cs} \to A_t$, связывается с каждой переменной объекта $(x, t) \in \overline{\text{INSTANCE_VARIABLE}}(cs)$ таким образом, что для каждой пары классов (для структур нет наследования) (c, c') если $c' <_{isa} c$, c' наследует x и $o \in Ac'$, то $x^A_{ct}(o) = x^A_{c't}(o)$, т. е. каждая переменная объекта представляется функцией, так что каждый подкласс, наследующий эту переменную, наследует и часть функции, поэтому если рассматривать объект как объект базового класса, то результат не изменится.

- Алгебраическая константа $x^A_{cst} : A_t$ связывается с каждой переменной класса $\overline{\text{STATIC_VARIABLE}}(cs)$, т. е. статическая переменная представляется алгебраической константой.
- Частичная функция $elem^A_{mt} : A_m \times A_{int} \to A_t$ связывается с операцией elem в каждом типе массива $m \in ARRAY(t)$. Эта операция не определена на **null**. Кроме этого у массива есть еще много функций, свойств и т.д., так как в C# массив является специальным классом, однако получение элемента по индексу является его основной операцией.

3.5. Обновление состояний

Одно состояние может быть преобразовано в другое посредством *обновления состояния* — *обновление основы* или *обновление памяти*. Обновление памяти изменяет содержимое области памяти, отвечающей за хранение одной переменной, т. е. либо объекта, либо переменной объекта. Обновление основы изменяет основу, т. е. множество адресов, используемых программой. Так как переменные в C# удаляются только автоматически, то обновление основы является, по сути, созданием нового объекта либо объявлением новой переменной.

3.6. Программа

Поведение программы характеризуется множеством состояний, называемым *носителем* программы и являющимся подмножеством множества всех состояний. В каждом состоянии любой объявленной функции программы (например, свойству объекта структуры или методу класса) сопоставляется функция алгебры. Причем в разных состояниях эти функции могут быть различны, так как могут зависеть от глобальных, по отношению к ним, параметров (например, метод объекта может зависеть от переменных объекта). Функция всегда вырабатывает пару — алгебру (возможно ту же, в которой она вызывается) и значение (элемент этой алгебры). На носитель накладывается одно условие: любая константа должна быть такой же в любом состоянии из носителя.

СПИСОК ЛИТЕРАТУРЫ

1. Borger E., Fruja G., Gervasi V., Stark R. F. A High-Level Modular Definition of the Semantics of C# // Theor. Comput. Sci. — 2005. — Vol. 336, N 2–3. — P. 343–365.
2. Jacobs B., Poll E. Coalgebras and monads in the semantics of Java // Theor. Comput. Sci. — 2003. — Vol. 291, N 3. — P. 329–349.
3. Borger E., Schulte W. A Programmer Friendly Modular Definition of the Semantics of Java // Lect. Notes Comp. Sci. — 1999. — Vol. 1523. — P. 353–404.
4. Замулин А. В. Формальная модель Java-программы, основанная на машинах абстрактных состояний // Программирование. — 2003. — N 3.
5. Замулин А. В. Алгебраическая семантика императивного языка программирования // Программирование. — 2003. — N 6.
6. C# Language Specification. Standard ECMA-334, 2005. — http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-334.pdf

СОДЕРЖАНИЕ

Предисловие 5
Андреева М.В. Временные структуры конфигураций: поведенческие эквивалентности и детализация действий 16
Батура Я.Н. Человеко-машинная модель языка мышления 29
Белоглазов Д.М. Обнаружение взаимодействия функциональностей в телефонных сетях с помощью раскрашенных сетей Петри 38
Ботоева Е.Ю. Двух- и трехмерная визуализации множества решений в системе UniCalc 48
Бражник С.А. Формальная модель диаграммы классов языка UML.......... 60
Веретнов С.О. Трансляция языка выполнимых спецификаций распределенных систем SDL в язык выполнимых спецификаций REAL... 76
Вольхина Н.К. Автоматическое восстановление бизнес-логики программ 90
Грибовская Н.С. Открытые морфизмы и временная тестовая эквивалентность для временных автоматных моделей 103
Демин А.В., Витяев Е. Е. Разработка модели адаптивного поведения анимата на основе семантического вероятностного вывода 121
Кальченко В.В. XML-алгебра для языка запросов XQuery 134
Пятков А.Б. Формальная модель основных понятий языка C# 150

CONTENTS

Preface 11
Andreeva M.V. Timed configuration structures: equivalence notions and action refinement.. 16
Batura Ya.N. Human-machine model of language of thinking.......................... 21
Beloglazov D.M. Detection of feature interaction in telephone networks using colored Petri nets 38
Botoeva E.Yu. 2D- and 3D-visualization of a solution set in the UniCalc system 48
Brazhnik S.A. Formal model of the UML class diagram 60
Veretnov S.A. Translation of a language of executable specifications of distributed systems SDL into a language of executable specifications REAL 76
Vol'khina N.K. Automatic recovery of program business logic 90
Gribovskaya N.S. Open maps and timed testing equivalence for timed automata models 103
Demin A.V., Vityaev E.E. The model of adaptive behavior based on the semantic probabilistic inference 121
Kal'chenko V.V. XML algebra for XQuery 134
Pyatkov A.B. Formal model of the basic concepts of the programming language C# 150

МОЛОДАЯ ИНФОРМАТИКА

СБОРНИК ТРУДОВ
АСПИРАНТОВ И МОЛОДЫХ УЧЕНЫХ

Выпуск 2

Под редакцией
к.ф.-м.н. И. С. Ануреева

Рукопись поступила в редакцию 20.11.06
Редактор З. В. Скок

Подписано в печать 28.12.06
Формат бумаги 60 × 84 1/16 Объем 9.7 уч.-изд.л., 10.6 п.л.
Тираж 75 экз.

Центр оперативной печати «Оригинал 2», г. Бердск, 49-а, оф. 7, тел./факс 8 (241) 5 38 77