МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ
МОСКОВСКИЙ ФИЗИКО-ТЕХНИЧЕСКИЙ ИНСТИТУТ
(ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ)

Кафедра информатики и вычислительной математики

Е. М. Лаврищева

ПРОГРАММНАЯ ИНЖЕНЕРИЯ

Тема 1. ТЕОРИЯ ПРОГРАММИРОВАНИЯ

Учебно-методическое пособие

МОСКВА
МФТИ
2016

УДК 681.3.06

Рецензент
Член-корреспондент РАН, доктор физико-математических наук,
профессор *И.Б. Петров*

Лаврищева, Е.М.

Программная инженерия. Тема 1. Теория программирования: учебно-методическое пособие. – М.: МФТИ, 2016. – 48 с.

Представлены школы по теории программирования (А.А. Ляпунова, Ю.И. Янова, А.П. Ершова, В.М. Глушкова, Е.Л. Ющенко, Г.Е. Цейтлина, В.Н. Редька и др.) на первых ЭВМ. Дана характеристика теории схем программ и автоматов, алгоритмического, алгебраического и синтезирующего программирования. Рассмотрены подходы к формальной спецификации программ и доказательства их правильности. Дана теория композиции и сборки модулей в сложные системы.

Предназначено для преподавания студентам 1–3 курсов, обучающихся в области информатики, программной инженерии и компьютерных наук.

Учебное издание

Лаврищева Екатерина Михайловна

ПРОГРАММНАЯ ИНЖЕНЕРИЯ

Тема 1. ТЕОРИЯ ПРОГРАММИРОВАНИЯ

Учебно-методическое пособие

Редактор *В.А. Дружинина.* Корректор *Л.В. Себова*. Компьютерная верстка *Л.В. Себова*
Подписано в печать 15.09.2016. Формат 60 × 84 $^{1}/_{16}$. Усл. печ. л. 3,0. Уч.-изд. л. 2,8.
Тираж 50 экз. Заказ № 383.

Федеральное государственное автономное образовательное
учреждение высшего образования «Московский физико-технический институт
(государственный университет)»
141700, Московская обл., г. Долгопрудный, Институтский пер., 9
Тел. (495) 408-58-22. E-mail: rio@mipt.ru

Отдел оперативной полиграфии «Физтех-полиграф»
141700, Московская обл., г. Долгопрудный, Институтский пер., 9
Тел. (495) 408-84-30. E-mail: polygraph@mipt.ru

ВВЕДЕНИЕ

Более двадцати лет назад А.П. Ершов писал, что теоретическое программирование является разделом математической науки, объектом изучения которой является абстрактная программа, выраженная логической структурой и информацией, подлежащих выполнению на компьютере. Теория программирования основывается на математических дисциплинах (логика, алгебра, комбинаторика) и отражает математический метод мышления специалиста при проведении анализа предметной области (ПрО), осмыслении постановок задач, описании программ для получения на машине математического результата. Теория программирования ориентирована на специалистов, обладающих математическими знаниями и способностью применять их к логике описания алгоритмов программ.

Этому взгляду А.П. Ершова соответствуют работы по теории программирования в СССР. Создана теория схем программ (А.А. Ляпунов, Ю.И. Янов и А.П. Ершов), которая представлена в книге А.П. Ершова «Введение в теорию программирования» (1977) [1–3]. В ней дано описание концепции теории схем программ, подходов к представлению программ в виде схем, доказательства правильности задания схемы, преобразования схемы к машинному представлению и др. Теоретическая схема программы описывалась первоначально в машинных языках, а затем в языках программирования (ЯП).

Следующим видом теории программирования является теория алгебраического и алгоритмического программирования (В.М. Глушкова), основанная на алгебраическом математическом аппарате для задания операций над элементами программ. Первым таким языком был Адресный язык (Е.Л. Ющенко), содержащий математические операции, задаваемые для представления данных и функций программ (1962) [4], а также определен принцип адресности, программного управления и средства манипулирования адресами второго ранга. Практически этот язык использовался для описания вычислительных программ и трансляторов (для машин «Днепр», «Проминь» и др.).

Развитием концепции Адресного языка является алгебраический математический аппарат, позволивший описывать математические задачи с применением алгебраических и адресных операций. В результате была создана монография «Алгебра. Языки. Программирование» (В.М. Глушков, Г.Е. Цейтлин, Е.Л. Ющенко, 1974) [5], которая была первым теоретическим изданием по теории программирования в СССР. Она была переведена на языки (англ., нем.) и переиздана в 1982 году. Алгебраическая теория получила практическое внедрение в языке «Аналитик» серии Машин

Инженерных Расчетов – «МИР 1, 2, 3» [6], который позволял решать инженерные задачи, применяя математические методы алгебраического вычисления таких задач.

Дальнейшим развитием теории схем программ была алгебра алгоритмики Г.Е. Цейтлина [7]. В ней алгоритм представлялся графовыми схемами и описывался с помощью математических операций. К ним относятся: операции над множествами, а также суперпозиция, свертка, развертка и др.

Теория программирования развивалась и за рубежом. Широко известна была переводная книга Д. Гриса «Наука программирования» (М.: Мир, 1984). Она содержала описание методов конкретизирующего и синтезирующего программирования разных программ обработки данных. Д.Е. Кнут «Искусство программирования» издал 4 тома (1968 – 1-й том, 1973 – 2-й и 3-й тома, 1981 – переизданные 2-й и 3-й тома и 2005 – тома 4, 4А и 4В). В них он дает описание понятия программы, операций сортировки и поиска информации, комбинаторику, алгебру, арифметику, описание трансляторов средствами языков типа Ассемблера. Дал описание компьютера для нового тысячелетия и др. Известна в СССР работа В. Турского «Методология программирования» (1980). В ней он описывает операции (последовательность, поиск и выбор) и подход к конструированию из них модулей и программ. Дается описание интерфейсов программ и средств расслоения программного продукта, подобно теории расслоения Л. Фуксмана (РГУ) [8–10].

В 60-годах прошлого столетия страну наводнили языки программирования (АЛГОЛ-60, КОБОЛ, ФОРТРАН и др.) как средства описания вычислительных и прикладных программ. Многие программисты стали их использовать при описании программ – вычислительных, экономических и др. Кроме того, в СССР развернулись работы по созданию трансляторов с этих ЯП. К трансляторам с языка АЛГОЛ относятся: ТА1 (Лавров С.С.), ТА2 (М.Р. Шура-Бура), ТА3 (А.П. Ершов), ТА4 (Е.М. Лаврищева). В результате начала развиваться технология программирования. В ней создана теория анализа языков программирования для каждого транслятора.

В практике сформировалось синтезирующее программирование [11]. Теория синтеза и система его поддержки разработана в системе ПРИЗ (Э.Х. Тыугу). Основу системы составляет модель постановки задачи ПрО и операции синтеза программ для их решения. Опубликованы монография «Инструментальная система программирования на ЕС ЭВМ (ПРИЗ)» М.И. Кахро и Э.Х. Тыугу (1981) и «Концептуальное программирование» Э.Х. Тыугу (1984) [12, 13].

В это время появились работы по математической формальной спецификации (VDM, Z, RSL, RAISE). D. Бернер [14] является главным авто-

ром языка VDM, который описывается ниже. Важное место занимает теория фундаментальных типов данных (Дейкстра, Хоар, Агафонов и др.) [15–17].

В результате сформировалась теория построения и доказательства правильности программ в рамках синтезирующего, сборочного и конкретизирующего программирования. А.П. Ершов на Всесоюзной конференции «Технология программирования» (1986) определил пути их развития до 2000 г. [18].

В данном учебно-методическом пособии содержится описание теории программирования в начальный период появления первых вычислительных машин. В нем более углубленно и с теоретической точки зрения рассматриваются первые теоретические методы программирования, обеспечения их правильности и качества работы.

Тема 1. ТЕОРИЯ ПРОГРАММИРОВАНИЯ

1.1. Теоретическое программирование

Первые школы программирования шли по пути создания теории схем программ, синтеза, композиции и сборки программ, а также алгоритмических и алгебраических методов программирования. В рамках этих школ определены и теоретически обоснованы формальные подходы к конструированию логических схем программ, методов их описания и реализации на ЭВМ.

Теоретически и практически программисты с математическим образованием развивали отдельные направления в программировании, объясняя некоторые закономерности в структуре программ и их определении с различных точек зрения: аппарата функций (функциональное программирование), композиций (композиционное программирование и др.); теории модульного программирования и сборки программ.

Первые работы в области теории программирования связаны с понятием схем программ, синтаксиса и семантики ЯП, моделей вычислений последовательных программ (операторных программ) и рекурсивных программ, которые строились в рамках некоторой абстрактной алгебраической системы [1–3].

При этом использовался алгебраический математический аппарат для объяснения действий над элементами программ, выполнения математических операций над этими элементами и принципов обработки, исходя из базовых основ алгебры – алгебраическое программирование, алгоритмика, композиции и др. [19–29].

В данном разделе представлены теоретические методы программирования (логическое, алгебраическое, композиционное, экспликативное, ал-

гебраическое и др.), основанные на математических, алгебраических и логико-алгоритмических подходах построения и доказательства программ, в том числе определения интеллектуальных агентов [23–25].

1.1.1. *Теория схем программ раннего программирования*

Структура теории схем программ сложилась на базе основополагающих работ А.А. Ляпунова и Ю.И. Янова [1–3], которые представили простейшую модель операторных схем с сигнатурой одноместных операций с одной переменной.

Схема программы – это конструктивный объект, задающий структуру программы с использованием формальных символов и операторов в виде конечного ориентированного графа переходов с одной входной и одной выходной вершиной. В графе имеется одна вершина – преобразователь и две вершины – распознаватели. Основным понятием теории схем является *понятие функциональной эквивалентности*.

Две схемы функционально эквивалентны, если для любой интерпретации соответствующие программы вычисляют одинаковые функции. Каждому выполнению программы сопоставляется протокол, отражающий порядок выполнения базовых операций.

Если известны значения истинности предикатов, входящих в программу, то протокол строится однозначно и не требует интерпретации базовых операций. Если выбирается значение истинности предикатов произвольно, то для схемы программы создается некоторое множество формальных протоколов, называемое ее *детерминантом*.

Две схемы формально эквивалентны, если их детерминанты совпадают. Формальная эквивалентность корректна, если из нее следует функциональная эквивалентность. Поскольку детерминант строится комбинаторно на основе произвольного выбора операций из конечного множества, он образует формальный язык, который воспринимается некоторым автоматом. Дается определение протокола схем программ, которые приводят к разрешимым детерминантам. Для таких детерминантов решается вопрос о поиске полной системы преобразований схем программ, при которой любые две формально эквивалентные схемы переводимы одна в другую.

Протоколом схемы Янова является последовательность выполняемых операций, перемежаемых значениями предикатов [6].

Автомат, воспринимающий детерминант, оказывается конечным автоматом, а формальная эквивалентность разрешима и совпадает с функциональной. Для схем Янова определена полная система преобразований.

Для общей модели операторных схем определена функциональная эквивалентность. Ю. Янов определил форму протокола с логико-термальной историей, приводящей к разрешимому детерминанту.

Этот протокол фиксирует последовательность выполнения и значения предикатов схемы и для каждого аргумента предиката указывается функциональный терм, который вычислял бы значение данного аргумента при выполнении программы.

Автомат, воспринимающий такой детерминант, является двухленточным конечным автоматом и для него определена полная система преобразований.

Существенное место в теории схем программ занимают вопросы перевода схем программ из одной вычислительной модели в другую. Операторные схемы эффективно переводятся в рекурсивные схемы в той же сигнатуре. Однако обратная трансляция невозможна, т.к. выполнение рекурсивной программы требует большого числа ячеек памяти.

Детерминант рекурсивной схемы может быть задан контекстно-свободным языком. Однако вопрос о разрешимости соответствующей формальной эквивалентности остался открытым.

Теория схем программ занималась изучением отдельных классов схем программ с целью выделения случаев разрешимых эквивалентностей.

1.1.2. *Логическое программирование. Теория программ*

В Logic или Logical programming программа представляет собой теорию и некоторое формальное *утверждение*, которое необходимо доказать. Теория задается с помощью *аксиом* и *правил вывода*. Утверждение, которое требуется доказать, вводится в программу в качестве *цели*. Работа программы состоит в поиске доказательства предложенного утверждения в имеющейся базе знаний, представляющей собой совокупность фактов и правил [19].

Разница между вводом и выводом условна. Можно указать желаемый вывод и получить результат. Другое важное отличие – недетерминизм логических языков. Результат необязательно определяется однозначно, поскольку произвольное целевое утверждение может быть доказано неединственным образом. Система, реализующая язык Prolog, предлагает возобновить доказательство утверждения по-другому. При доказательстве утверждений проводится сопоставление двух произвольных термов по методу резолюций.

Логика схемы программы. Программа A частично правильна относительно входного условия P и выходного условия Q, т.е. $P\{A\}(Q)$ правильно, если P истинно для входных значений переменных, A завершает работу и Q истинно для выходных значений переменных.

При этом P называют *предусловием*, а Q – *постусловием* программы A. Программа A правильна (обозначение $P < A > Q$), если A частично пра-

вильна относительно P и Q, а A завершает работу для входных значений переменных, удовлетворяющих условию P.

Для доказательства частичной правильности последовательных программ используется метод Флойда, сущность которого состоит в следующем. На схеме программы выбираются контрольные точки так, чтобы любой циклический путь проходил по крайней мере через одну точку. Контрольные точки также связываются с входом и выходом из точек схемы.

С каждой контрольной точкой ассоциируется специальное условие (индуктивное утверждение или инвариант цикла), которое истинно при каждом переходе через эту точку. С входной и выходной точками ассоциируются входное и выходное условия. Затем каждому пути программы между двумя соседними контрольными точками сопоставляется условие правильности. Выполнимость всех условий правильности гарантирует частичную правильность программы.

Один из способов доказательства завершения работы программ состоит во введении в программу дополнительных счетчиков и установлении ограниченности этих счетчиков на выходе программы в процессе доказательства частичной правильности.

Аксиоматическая семантика языков программирования задается посредством конечной аксиоматической системы (Hoare C.A.R. Prof of correctness of data representation // Acta Informatica. 1972. V. 1(4). P. 271–281).

Логика Хоара состоит из аксиом и правил вывода, в которой в качестве теорем выводимы утверждения о частичной правильности программ. Например, для оператора присваивания схема аксиом имеет вид

$$P\ (x \leftarrow c)\ \{x := c\}\ P,\ P\ (x \leftarrow c),$$

где $P\ (x \leftarrow c)$ означает результат подстановки в P выражение c вместо всех вхождений переменной x, а для оператора цикла типа *while* правило вывода имеет вид

$$(P \wedge R)\ \{A\}\ P \vdash P\ \{\text{while } R \text{ do } A\}\ (P \wedge \neg R),$$

то есть P является инвариантом цикла.

Пусть рассматривается логика Хоара, в которой в качестве языка для записи условий взят язык арифметики 1-го порядка.

Утверждение о частичной правильности программы называется *выводимым*, если оно выводимо в этой логике посредством добавления истинных формул арифметики, и называется *истинным*, если оно истинно по отношению к операционной (или денотационной) семантике программ.

Логика Хоара называется *состоятельной*, если каждое выводимое в ней утверждение истинно, и *полной*, если каждое истинное утверждение выводимо в ней. Состоятельная и полная логика Хоара построена, в частности, для ЯП, содержащего простые переменные и операторы присваи-

вания, составной, условный оператор, а также циклы и процедуры. Важным обобщением логики Хоара является так называемая *алгоритмическая* (*или динамическая*) *логика*.

Пусть $P\{A\}$ представлено в виде $P \subset [A]Q$, где $[A]Q$ – слабое предусловие, для которого справедливо утверждение о частичной правильности программы A с постусловием Q.

Аналогично, для случая тотальной правильности пусть $P < A > Q$ представлено в виде $P \subset A > Q$. Формулы алгоритмической логики строятся из формул базового логического языка и программ, задаваемых с помощью булевых операций, кванторов, а также операций вида

$[A] < Q$, $< A > Q$.

В алгоритмической логике задаются утверждения о программах, например, эквивалентность. Аналогично логике Хоара для алгоритмической логики построена состоятельная и полная конечная аксиоматическая система и в случае ЯП допускает определение недетерминированных программ.

Для доказательства утверждений о рекурсивных программах часто используется специальная индукция, связанная с определением теории.

Пусть рекурсивная программа задается одним уравнением $f = t\ (f)$, а ее денотационная семантика – fn.

При естественных предположениях справедлив следующий принцип индукции: если формула $P(W)$ истинна и из $P(t^i(W))$ следует $P(t^{i+1}(W))$, то выполняется $P(fn)$ (здесь W – нигде не определенная функция). Для описания денотационной семантики ЯП используется задание области данных в виде так называемых *полных решеток Скотта*.

Задачу синтеза программы можно формализовать как задачу построения доказательства теоремы $\forall x\ \exists\ y\ \Pi\ (x, y)$ и последующего извлечения программы из этого доказательства. Построен алгоритм, который по доказательству в интуиционистской логике дает программу на языке Алгол-68. Если доказательство использует правило индукции, то ему соответствует в программе цикл вида for to ... do.

Извлекаемая программа будет приемлемой сложности, если использовать предположение, что она строится из стандартных функций и предикатов, свойства которых описаны аксиомами специального вида.

1.1.3. *Теория формальной спецификации программ*

К формальным методам спецификации относятся: Венский метод VDM D. Biorner, Z-метод (I.R. Abrial, B. Meyer), RAISE и др. [14–17]. Эти методы начали использоваться в реальных проектах, а также в университетских и академических организациях. Язык VDM служит способом спецификации программ и данных с помощью математической символики и следующих типов данных:

X – натуральные числа с нулем,

N – натуральные числа без нуля,

Int – целые числа,

Bool – булевы,

Qout – строки символов,

Token – знаки и специальные обозначения операций.

Функция в VDM задает определение свойств структур данных и операций над ними, аппликативно или императивно. В первом случае функция специфицируется через комбинацию других функций и базовых операций (через выражения), что соответствует синониму *функциональный*. Во втором случае значение определяется описанием алгоритма, что соответствует синониму *алгоритмический*. Например, спецификация функции вычисления минимального значения из двух переменных имеет вид: min N_1 N_2 $\rightarrow$ N_3. Алгоритмическое описание значения этой функции имеет вид: min (x, y) = if $x < y$ then x else y.

Объекты в языке VDM. Элементы данных, которыми оперируют функции, могут образовывать множества, деревья, последовательности, отображения, а также формировать новые более крупные объекты.

Множество может быть конечное и обозначаться X-set. При работе с множеством используются операции $\in$, $\subseteq$, $\cup$, $\cap$ и др. Они используются для проверки правильности задания этих операций. Пример: $x \in A$ будет корректным только тогда, когда A является подмножеством из множества, которому принадлежит x.

Дистрибутивное объединение подмножеств имеет, например, вид:

union {(1, 2), (0, 2), (3, 1) } = (0, 1, 2, 3).

Списки (*последовательности*) – это цепочки элементов одинакового типа из множества X. Операция **len** задает длину списка, а **inds** – номер элемента списка. Например, **inds** lst = ($i \in X$ [$f \leq i \leq$ len]).

К операциям относятся: взятие первого (головы) элемента списка – **hd** и остатка (хвоста) после удаления первого элемента из списка – **tl**.

Например, **hd** (a, b, c, d) = (a), **tl** (a, b, c, d) = (b, c, d).

Могут использоваться также операция *конкатенация* (соединение двух списков) и операция *дистрибутивная конкатенация*.

Дерево – это конструкция **mk**, позволяющая объединять в комплекс объекты разной природы (последовательности, множества и отображения). В деревьях могут использоваться также конструктор для обозначения составных объектов и деструктор для именования констант, вносимых в ранее определенный составной объект. Например, let **mk –** время (h, m) = t **tin** определяет значение h = 10, а m = 30.

Отображение – это конструкция **map**, позволяющая создавать абстрактную таблицу из двух столбцов: ключей и значений. Все объекты

таблицы принадлежат одному типу данных – множеству. При этом операция dom строит множество ключей, а rng – множество всех его значений. Есть еще такие операции: исключить строку, слить две таблицы и др.

При спецификации программ средствами VDM задаются пред- и постусловия, аксиомы и утверждения, необходимые для проведения доказательства правильности специфицированной программы. *Предусловие* – это предикат с операцией, к которой будет обращаться программа после получения ее начального состояния или в случае нерегулярной ситуации.

Утверждение задает описание операций проверки правильности программы в разных ее точках. Операторы программы изменяют ее состояние, а операции утверждений извлекают информацию из нее после изменения состояния. На основе этой информации устанавливается правильность или неправильность выполнения операции.

Постусловие – это предикат, который является истинным после выполнения предусловия, завершения текущей операции и выполнения инвариантных свойств программы.

Метод VDM ориентирован на пошаговую детализацию спецификации программ. На первом уровне строится грубая спецификация – модель программы в языке VDM, которая постепенно уточняется, пока не получится окончательный текст в ЯП. Реальная разработка спецификации производится итерационно, на первом уровне она служит для проверки только свойств модели программы при заданных ограничениях и независимо от среды. Далее уточняется и расширяется спецификация и набор формальных утверждений. И так до тех пор, пока окончательно не будет завершен процесс спецификации программы и проведено доказательство.

Доказательство инвариантных свойств программ проводится автоматизированным способом с помощью специально созданных инструментальных средств поддержки VDM-языка.

1.1.4. *Алгебраическая теория программ*

Примером применения алгебраических методов может служить проблема эквивалентности дискретных преобразователей теории Глушкова [20–25], в которую естественно вкладываются операторные схемы программ.

Пусть χ – конечный автомат Мили с входным алфавитом X и выходным алфавитом Y и с заданными начальным и заключительным состояниями.

Пусть G – полугруппа с множеством образующих Y единицей e и рассматривается автомат Мура G_m (бесконечный) с множествами состояний G, входов Y, выходов X, начальным состоянием e, функцией выхода $m(g)$ и функцией перехода $q(g, y) = gy$.

Автомат χ, работающий совместно с G_m, называется *дискретным преобразователем*. Если этот автомат χ в качестве входа воспринимает выход G_m, то в качестве выхода определяется χ. Выходом χ является состояние G_m в момент остановки χ. Дискретные преобразователи эквивалентны относительно полугруппы G, если для каждого отображения m из G в X оба не имеют остановки при работе G_m либо оба останавливаются и имеют одинаковый выход.

Проблема эквивалентности дискретных преобразователей разрешима относительно полугруппы с левым сокращением и неразрешимой единицей, в которой разрешима проблема тождества слов. Описаны также все разрешимые и неразрешимые случаи эквивалентности дискретных преобразователей относительно коммутативной полугруппы.

Теория оказывает свое влияние на практику, прежде всего, как концептуальная основа схем программ и как техника доказательства программ в формальных ЯП. Свойства абстрактных моделей вычислений используются для уточнения семантики ЯП и обоснования различных операций с программами.

Сюда примыкает программирование констрайнами (constraint programming). В нем постановка задачи – это конечный набор переменных $V = \{v_1, ..., v_n\}$, соответствующих конечному (перечислимому) множеству значений $DV = \{dv_1, ..., dv_n\}$, и набор ограничений $C = \{c_1, ..., c_m\}$. Система ограничений (constraint) может включать в себя уравнения, неравенства, логические функции, а также любые допустимые формальные конструкции, связывающие переменные из набора V. Решение задачи состоит в построении набора значений переменных, удовлетворяющего всем ограничениям c_i, где $i = 1, \ldots, m$.

Система программирования в ограничениях представляет собой интерпретатор языка Prolog со встроенным механизмом для решения определенного класса задач в ограничениях. Она называется *Constraint Logic Programming* или CLP, или CLP(X), где X указывает на класс решаемых задач. Например, CLP(B) означает возможность решать уравнения с булевыми переменными, CLP(Q) – уравнения в рациональных числах, а CLP(R) – в вещественных. Наиболее популярны решатели задач на конечных множествах целых чисел CLP(FD).

С точки зрения семантики программирование в ограничениях отличается от традиционного логического программирования в первую очередь тем, что исполнение программы рассматривается не как доказательство утверждения, а как нахождение значений переменных. При этом внутренний порядок решения не имеет значения для выполнения отдельных ограничений, и система программирования в ограничениях, как правило,

стремится оптимизировать порядок доказательства утверждений с целью минимизации отката в случае неуспеха.

Любая логическая программа может быть преобразована в эквивалентную программу с ограничениями и наоборот. Иногда логические программы специально конвертируют в программы с ограничениями, поскольку выполнить решение задачи с ограничениями бывает проще, чем произвести логический вывод программы.

Наиболее популярными языками программирования в ограничениях являются языки, базирующиеся на Prolog: B-Prolog, CHIP V5, Ciao Prolog, ECLIPSe, GNU Prolog и др.

1.1.5. *Теория алгоритмики программ*

На протяжении многих лет Г.Е. Цейтлин занимался разработкой алгебро-алгоритмического программирования, основанного на теории алгоритмов, представляемых блок-схемами, графами и математическими формулами аналитического преобразования программ. Построение и исследование алгебры алгоритмов впервые предложил академик В.М. Глушков в рамках проектирования логических структур ЭВМ и назвал *системой алгоритмических алгебр* (САА). Эту теорию Цейтлин положил в основу обобщенной теории структурных схем алгоритмов и программ, назвав ее *алгоритмикой* [5, 7].

Объекты алгоритмики – модели алгоритмов и программ, представляемые в виде схем. Алгоритмика базируется на компьютерной алгебре, логике и используется для проектирования алгоритмов прикладных задач. Построенные схемы алгоритмов описываются с помощью ЯП и реализуются соответствующими системами программирования в машинное представление. В рамках алгоритмики разработаны специальные инструментальные средства реализации алгоритмов, которые базируются на идеях современных ООП и метода моделирования UML.

Таким образом, приведен полный цикл работ по практическому применению теории алгоритмики для задач, начиная с их постановки, формирования требований, разработки алгоритмов и кончая получением программ решения этих задач.

Алгебра алгоритмов. Под алгеброй алгоритмов $AA = \{A, \Omega\}$ понимается основа A и сигнатура Ω операций над элементами алгебры. С помощью операций сигнатуры может быть получен произвольный элемент $q \in AA$, который называется *системой образующих алгебр*. Если из этой системы не может быть исключен ни один элемент без нарушения их свойств, то такая система образующих называется *базисом алгебры* [10].

Операции алгебры удовлетворяют следующим аксиоматическим законам: *ассоциативности, коммутативности, идемпотентности, закону ис-*

ключения третьего и др. Алгебра, которой удовлетворяют перечисленные операции, называется *булевой*.

В алгебре алгоритмов используется алгебра множеств, элементами которой являются множества и операции над множествами (объединение, пересечение, дополнение, универсум и др.).

Основными объектами алгебры алгоритмики являются схемы алгоритмов и их суперпозиции, т.е. подстановка одних схем в другие. С подстановкой связана развертка, которая соответствует нисходящему процессу проектирования алгоритмов, и свертка, т.е. переход к более высокому уровню спецификации алгоритма. Схемы алгоритмов соответствуют конструкциям структурного программирования.

Последовательное выполнение операторов A и B записывается в виде композиции $A * B$; альтернативное выполнение операторов A и B ($f(A, B)$) означает, если u истинно, то выполняется A, иначе – B; цикл (u (A, B)) выполняется, пока не станет истинным условие u (u – логическая переменная).

С помощью этих элементарных конструкций строится более сложная схема П алгоритма:

$\Pi ::= \{(u_1)\, A_1\}$,

$A_1 ::= \{(u_2)\, A_2 * D\}$,

$A_2 ::= A_3 * C$,

$A_3 ::= \{(u)\, A, B\}$, $u ::= u_2 \wedge u_1$.

Проведя суперпозицию и свертки приведенной схемы алгоритма П, получаем формулу

$\Pi ::= \{(u_1)\, (u_2)\, (u_2 \wedge u_1)\, A, B) * C * D)\}$.

Важным результатом работы является эквивалентное представление и толкование известных алгебр (Дейкстры, Янова, Глушкова) с помощью алгебры алгоритмики.

Алгебра Дейкстры. АД = {*CAA*, *L*(2), СИГН} – двухосновная алгебра, основными элементами которой являются множество *CAA* операторов, представленных структурными блок-схемами, множество *L*(2) булевых функций в сигнатуре СИГН, в которую входят операции дизъюнкции, конъюнкции и отрицания, принимающие значения из *L*(2). С помощью специально разработанных механизмов преобразования АД в алгебре алгоритмики установлена связь между альтернативой и циклом, т.е. $\{(u)\, A\} = \{(u)\, E, A * \{(u)\, A\}\}$, произвольные операторы представлены суперпозицией основных операций и констант. Операция фильтрации $\Phi(u) = = \{(u)\, E, N\}$ в АД представлена суперпозицией тождественного E, неопределенного N оператора и альтернативы алгебры алгоритмики, где N – фильтр разрешения выполнения операций вычислений. Оператор цикла **while do** также

представлен суперпозицией операций композиции и цикла в алгебре алгоритмики.

Алгебра схем Янова (АЯ) включает в себя операции построения неструктурных логических схем программ. Схема Янова состоит из предикатных символов множества $P\{p_1, p_2, ...\}$, операторных символов множества $A\{a_1, a_2, ...\}$ и графа переходов. Оператором в данной алгебре является пара $A\{p\}$, состоящая из символов множества A и множества предикатных символов. Граф перехода представляет собой ориентированный граф, в вершинах которого располагаются преобразователи, распознаватели и один оператор останова. Дуги графа задаются стрелками и помечаются знаками + и –. Преобразователь имеет один преемник, а распознаватель – два. Каждый распознаватель включает в себя условие выполнения схемы, а преобразователь представляет собой операторы, состоящие из логических переменных, принадлежащих множеству $\{p_1, p_2, ...\}$.

Каждая созданная схема АЯ отличается большой сложностью, требует серьезного преобразования при переходе к представлению программы в виде соответствующей последовательности действий, условий перехода и безусловного перехода. В работе [5, 7] разработана теория интерпретации схем Янова и доказательство эквивалентности двух операторных схем, исходя из особенностей алгебры алгоритмики.

Для представления схемы Янова аппаратом алгебры алгоритмики в сигнатуру операций АЯ вводятся композиции $A * B$ и оператор условного перехода, который в зависимости от условия выполняет переход к следующим операторам или к оператору, помеченному меткой (типа goto). Условный переход трактуется как бинарная операция $\Pi (u, F)$, которая зависит от условия u и разметок схемы F. Кроме того, производится замена цикла типа while do. В результате выполнения бинарных операций получается новая схема F', в которой установлена $\Pi(u)$ вместо метки, и булевы операции конъюнкции и отрицания. Эквивалентность выполненных операций преобразования обеспечивает правильность неструктурного представления.

Система алгебр Глушкова (АГ). Она имеет вид: АГ = {ОП, УС, СИГН}, где ОП и УС – множество операторов суперпозиции, входящих в сигнатуру СИГН, и логических условий, определенных на информационном множестве ИМ, СИГН = {СИГНад $\cup$ Прогн.}, где СИГНад – сигнатура операций Дейкстры, Прогн. – операция прогнозирования. Сигнатура системы алгоритмических алгебр (САА) включает операции: алгебры АД, обобщенной трехзначной булевой операции, а также операцию прогнозирования (левое умножение условия на оператор $u = (A * u')$. Эта операция порождает предикат u = УС, такой, что $u(m) = u'(m')$, $m' = A(m)$, $A \in$ ОП. ИМ – множество обрабатываемых данных и определения операций из множеств ОП и УС. Сущ-

ность операции прогнозирования состоит в проверке условия u в состоянии m оператора A и определения условия u', вычисленного в состоянии m' после выполнения оператора A.

Данная алгебра ориентирована на аналитическую форму представления алгоритмов и оптимизацию алгоритмов по выбранным критериям.

Алгебра булевых функций и связанные с ней теоремы о функциональной полноте и проблемы минимизации булевых функций также сведены к алгебре алгоритмики. Этот специальный процесс отличается громоздкостью и не приводится.

Алгебра алгоритмики и прикладные подалгебры. Алгебра алгоритимки пополнена двухуровневой алгебраической системой и механизмами абстрактного описания данных (классами алгоритмов).

Под многоосновной алгоритмической системой (МАС) понимается система $S = \{\{D_i / i \in I\}, СИГН_0, СИГН_n\}$, где D_i – основы или сорта, $СИГН_0$, $СИГН_n$ – совокупности операций и предикатов, определенных на D_i. Если они пусты, то определяются многоосновные модели – алгебры. Если сорта интерпретируются как множество обрабатываемых данных, то МАС представляет собой концепцию абстрактных типов данных (АТД) в виде подалгебры, широко используемой в ООП. Этим установлена связь с современными тенденциями развития программирования.

1.1.6. *Теория синтезирующего программирования*

Теория синтеза состоит в построении программы из условия задачи и метода ее решения. Условие задачи и метод ее решения переплетены таким образом, что это выглядит как перечисление объектов, дополненное описанием свойств этих объектов в виде разного рода соотношений или условий. В этом случае метод решения должен быть выбран или найден с помощью анализа перечисленных свойств и воплощен в тексте программы. С формальной точки зрения условие задачи записывается в виде совокупности логических формул, в которые входят символы известных и неизвестных величин, операций и функций. Такая совокупность формул получила название *спецификации* задачи. В синтезе программы по спецификациям различают два подхода: логический и аналитический (или трансформационный) [8–13].

При логическом подходе спецификация трактуется как формулировка теоремы, утверждающей существование решения задачи. Синтез программы состоит в поиске доказательства этой теоремы существования в некотором конструктивном логическом исчислении. Если такое доказательство удается построить, то существуют формальные правила, позволяющие преобразовать доказательство в программу, находящую решение задачи.

При аналитическом подходе спецификация трактуется как уравнение для программы. Уравнение можно подвергать символическим преобразованиям, при которых символ неизвестной программы «рассыпается» на систему вспомогательных неизвестных, а они, в свою очередь, заменяются либо другими неизвестными, либо конкретными программными конструкциями. Таким образом, по мере преобразований синтезируемая программа постепенно «прорастает» в тексте спецификации, превращая ее в законченный программный текст. Правильность выполнения символических преобразований может формально проверяться на каждом шагу.

При обоих подходах синтез программы из спецификаций является творческой задачей: в первом случае надо найти доказательство, во втором – правильное направление преобразований. На практике применяется комбинация обоих подходов.

Если говорить менее формально, то синтез программы – это некоторый способ манипулирования знаниями: специальным знанием, воплощенным в условии задачи, предметным, характеризующим область, в которой возникла задача, и универсальным знанием, отражающим общематематические закономерности и правила доказательного рассуждения. В целом такое словоупотребление оказалось методологически весьма плодотворным, поскольку позволило объединить универсальную строгость математической логики с разнообразием предметных областей, проницаемость человеческой интуиции с безошибочной пунктуальностью машинной обработки.

Основу синтезирующего программирования составляет преобразование, которое опирается на ряд вспомогательных логических утверждений, которые при этом доказываются. Применяется разнообразная формальная манипуляционная техника, при которой используются разные сведения, образующие определенную систему или, как говорят в программировании, базу знаний.

Вопросами синтеза программ в отечественной практике занимался С.С. Лавров, теория синтеза представлена им в статье «Синтез программ» [11]. Теория синтеза и система его поддержки разработана и в системе ПРИЗ (Э.Х. Тыугу). Основу этой системы составляет модель постановки задачи и операции синтеза программ для их решения. Она описана в монографии «Инструментальная система программирования на ЕС ЭВМ (ПРИЗ)» М.И. Кахро и Э.Х. Тыугу (1981) и «Концептуальном программировании» Э.Х. Тыугу (1984) [12, 13].

1.1.7. *Функциональное программирование*

Данное программирование (Functional programming) применяется для решения задач, трудно формулируемых в терминах последовательных операций – для задач, связанных с распознаванием образов, общением на естественном языке, реализацией экспертных систем, автоматизированным доказательством теорем, символьными вычислениями и др. [19].

Основу функционального программирования составляет теория лямбда-исчисления (А. Черча) и комбинаторная логика (М. Шенфинкеля и Х. Карри). Функциональная программа состоит из совокупности *определений функций*, представляющих собой вызовы других функций и предложений, управляющих последовательностью вызовов. Описание *зависимости* функций друг от друга (возможно, рекурсивной) фактически определяет способ решения задачи (без указания последовательности шагов). Рекурсия является фундаментальной основой построения семантики языков функционального программирования.

В общем случае функциональная программа не содержит оператора присваивания, вычисление любой функции не приводит ни к каким побочным эффектам, отличным от собственно вычисления ее *значения*. Разветвление вычислений основано на механизме обработки аргументов условного предложения, а циклические вычисления реализуются с помощью рекурсии.

Наиболее известны следующие функциональные ЯП:

- LISP (1958 год) и множество его потомков, наиболее современные из которых – Scheme и Common Lisp;
- ML (1979 год) и его наиболее известные диалекты – Standard ML и Objective CAML;
- Miranda (1985) дал развитие языку Haskell и др.

Языки функционального программирования в основном используются в научных (академических), а не в коммерческих приложениях, за исключением языка Erlang (для разработки высокопроизводительных приложений в телекоммуникации), языков J and K (для финансового анализа), а также таких доменных языков, как, например, XSLT.

1.1.8. *Автоматное программирование*

Это программирование основано на применении *теории конечных автоматов* для описания поведения программ. Автоматы задаются графами переходов для различения вершин, в которых вводится понятие *кодирование состояний*. Особенность автоматного программирования состоит в том, что графы переходов используются при *спецификации*, *проектировании*, *реализации*, *отладке*, *документировании* и *сопровождении* программ [31].

Программирование выполняется «через состояния», а не «через события и переменные», что позволяет лучше понять и специфицировать задачу и ее составные части. Переход от графового представления к текстовому осуществляется формально и изоморфно с применением конструкции switch (в языке С) или ее аналогов (в других языках). Поэтому стиль автоматного программирования часто называют «Switch-технологией».

В настоящее время этот стиль развивается в нескольких вариантах, различающихся как классом решаемых задач, так и типом вычислительных устройств, на которых осуществляется программирование. Известны, например, его варианты для систем логического управления, в которых события отсутствуют, входные и выходные воздействия являются двоичными переменными, а операционная система работает в режиме сканирования. Автоматный подход распространен и на событийные системы, называемые также *реактивными*. В них входные воздействия используют события в качестве выходных воздействий – произвольные процедуры, а в качестве операционных систем — любые операционные системы реального времени.

Для программирования событийных систем с применением автоматов используется процедурный подход, поэтому такой стиль программирования называется «программированием с явным выделением состояний». Известен также подход, основанный на совместном использовании объектного и автоматного стилей и называемый «объектно-ориентированным программированием с явным выделением состояний».

В контексте обеспечения качества применение автоматов проясняет *поведение* программы, а наличие хорошей проектной документации упрощает ее изменение путем рефакторинга программы.

1.2. Парадигмы программирования

Парадигма (от греч. παράδειγμα, «пример, модель, образец») – совокупность фундаментальных научных установок, представлений и терминов, принимаемая и разделяемая научным сообществом. Обеспечивает преемственность развития науки и научного творчества. Томас Кун называл парадигмами устоявшиеся системы научных взглядов, в рамках которых ведутся исследования и разработка [30].

Парадигма программирования – это совокупность идей и понятий (теорий, методов), определяющих стиль написания компьютерных программ. Этот термин Р.М. Флойд определил в своей работе «The Paradigms of Programming» (Communications of the ACM. 1969. V. 22(8). P. 455–460), Э. Дейкстра в книге «Дисциплина программирования» (М.: Мир, 1976) и Д. Грис в книге «Наука программирования». Он назвал парадигму как

способ концептуализации и определения организации вычислений для выполнения программы на компьютере [5–13].

Парадигма программирования предоставляет теорию, методы и средства программирования. Дадим классификацию парадигм:

- алгебраическое и логическое, объектно-ориентированное и агентное программирование [15, 32–24];
- синтезирующее, сборочное и конкретизирующее программирование [18, 28–30];
- машинно-ориентированное, трансформационное, высокопроизводительное/параллельное программирование) [31, 32] и другие.

Далее дается характеристика основных видов парадигм.

Согласно программы обучения программной инженерии SWEBOK (2001–2007) в учебные курсы обучения студентов включены курсы CS1011 «Programming fundamentals»), CS102I «The Object-Oriented Paradigm», а также курс событийного и параллельного (event-based programming) программирования на примере ЯП (C++, C#, Basic, Java, Python и др.).

Ниже рассматриваются парадигмы алгебраического, экспликативного, композиционного, модульного, сборочного программирования.

1.2.1. *Парадигма алгебраического программирования* (*АП*)

Данная парадигма [5–13] основывается на теории переписывания термов. В этой парадигме *термы* представляют данные, а системы *переписывающих правил*, выражаемых с помощью системы равенств, – алгоритмы вычислений. Элементарный шаг вычисления включает сопоставление с образцом, проверку условий и подстановку. Порядок выбора переписывающих правил и подтермов данного терма для сопоставления с левыми частями равенств определяется *стратегией* переписывания. По существу, стратегия определяет результат вычислений – терм – с точностью до эквивалентности исходному терму. Собственно, стратегия переписывания может быть описана в парадигме более низкого уровня, например процедурной или функциональной, что приводит к необходимости интеграции парадигм. В настоящее время используются специализированные структуры данных – *графовые термы* – для представления данных и знаний о предметных областях.

Данная парадигма развивается А.А. Летичевским в направлении интеллектуальных агентов и среды их функционирования с применением математического аппарата, в качестве которого используется понятие транзитивной системы [23–25]. Данный аппарат позволяет определить поведение систем и их эквивалентность. Элементами транзитивных систем в общем случае могут быть компоненты, программы и объекты, взаимодействующие друг с другом и со средой их существования. Эволюция

такой системы описывается с помощью истории функционирования систем, которая может быть конечной или бесконечной и включать в себя обзорную часть в виде последовательности действий и скрытую часть в виде последовательности состояний.

История функционирования включает в себя успешное завершение вычислений в среде транзитивной системы, тупиковое состояние, когда каждая из параллельно выполняющихся частей системы находится в состоянии ожидания событий, и, наконец, неопределенное состояние, возникающее при выполнении алгоритма, например, с бесконечными циклами.

Главное инвариантное состояние транзитивной системы – поведение системы, которое можно задать выражениями алгебру поведения $F(A)$ на множестве операций алгебры A. Две операции префиксинга $a \cdot u$ задает поведение u на операции a и недетерминированный выбор $u + v$ одного из двух поведений u и v, который является ассоциативным и коммутативным. Конечное поведение задается константами: Δ, $\perp$, 0, обозначающими соответственно состояния успешного завершения, неопределенного и тупикового. Алгебра поведения частично задается отношением $\leq$, для которого элемент $\perp$ – наименьший, а операции алгебры поведения – монотонные. Теоретически алгебра поведения $F(A)$ проверена путем доказательства теоремы о наименьшей неподвижной точке.

Транзитивные системы называются *бисимуляцийно эквивалентными*, если каждое состояние любой из них эквивалентно состоянию другой. На множестве поведений определяются операции, которые используются для построения программ агентов, а именно такие операции: последовательная композиция $(u; v)$ и параллельная $u \parallel v$.

Среда E, где находится объект, определяется как агент в алгебре действий A и функции погружения от двух аргументов $Ins\ (e, u) = e[u]$. Первый аргумент – это поведение среды, второй – поведение агента, который погружается в эту среду в заданном состоянии. Алгебра агента – это параметр среды. Значение функций погружения – это новое состояние одной и той же среды.

Базовым понятием является «действие», трансформирующее состояние агентов, поведение которых, в конце концов, изменяется.

Поведение агентов характеризуется состоянием с точностью до бисимуляции и, возможно, слабой эквивалентности. Каждый агент рассматривается как транзитивная система с действиями, определяющими недетерминированный выбор и последовательную композицию (т.е. примитивные и сложные действия).

Взаимодействие агентов может быть двух типов. Первый тип выражается через параллельную композицию агентов над той же самой областью действий и соответствующей комбинацией действий. Второй тип

выражается через функцию погружения агента в некоторую среду, и результатом трансформации является новая среда.

Язык действий *A* (Action) имеет синтаксис и семантику. Семантика – это функция, определяемая выражениями языка и ставящая в соответствие программным выражениям языка значения в некоторой семантической области. Разные семантические функции дают равные абстракции и свойства программ. Семантика может быть вычислительной и интерактивной. Доказано, что каждая алгебра действий является гомоморфным образом алгебры примитивных действий, когда все слагаемые разные, а представление однозначно с точностью до ассоциативности и коммутативности в детерминированном выборе. Установлено, что последовательная композиция – ассоциативная, а параллельная композиция – ассоциативная и коммутативная. Параллельная композиция раскладывается на комбинацию действий компонентов.

Агенты рассматриваются как значения транзитивных систем с точностью до бисимуляционной эквивалентности. Эквивалентность характеризуется в алгебре поведения непрерывной алгеброй с аппроксимацией и двумя операциями: недетерминированным выбором и префиксингом. Среда вводится как агент, куда погружена функция, имеет поведение типа *агент* и *среда*. Произвольные непрерывные функции могут быть использованы как функции погружения, а эти функции определены значениями логики переписывания. Трансформации поведения среды, которые определяются функциями погружения, составляют новый тип эквивалентности – эквивалентность погружения.

Создание новых методов программирования с введением агентов и сред позволяет интерпретировать элементы сложных программ как самостоятельно взаимодействующие объекты [21].

В АП интегрируется процедурное, функциональное и логическое программирование, используются специальные структуры данных – граф термов, который разрешает применять разные средства представления данных и знаний о ПрО в виде выражений многоосновной алгебры данных. Наибольшую актуальность имеют системы символьных вычислений, которые дают возможность работать с математическими объектами сложной иерархической структуры. Многие алгебраические структуры (группы, кольца, поля) являются иерархически-модулярными. Теория АП обеспечивает создание математической информационной среды с универсальными математическими конструкциями, вычислительными механизмами, учитывающими особенности разработки ПС и функционирования. АП является основой формирования нового вида программирования – *инсерционного*, обеспечивающего программирование систем на основе моделей

поведения агентов, транзитивных систем и бисимуляционной эквивалентности [24].

Инструменты. Система алгебраического программирования (АПС) основывается на типах системных объектов, базовых вычислительных механизмах и языковых конструкциях AL [25]. Она объединяет процедурный и алгебраический методы программирования, разрешает использовать не только канонические, но и другие системы уравнений. ПрО реализуется: алгебраическими программами (АП-модуль), алгебраическими модулями (А-модуль), интерпретаторами алгебраических модулей и т.п. А-модуль – это представление структуры данных, которые определяются в АП-модулях. Он наследует тип, начальное именование и имеет динамический характер, т.е. состояние, которое может изменяться во времени. Техника программирования в АПС базируется на технике переписывания терминов и используется при автоматизации доказательства теорем, символьных вычислениях, обработке алгебраических спецификаций. В АПС допускается модельно-имитационный класс, т.е. мониторинг данных, моделирование ситуации, условное прерывание, управление экспериментами и др. Дедуктивно-трансформационный класс АПС включает методы наблюдений, доказательства и повторного использования программ.

Таким образом, это программирование обобщает взгляд на программу как на алгебраическое *преобразование* множества состояний информационной среды. Преобразование поведения этой среды происходит в результате воздействия на ее взаимодействующих *агентов*. В отличие от агентного программирования, концентрирующего внимание на проблемах интеллектуализации агентов, программирование АП охватывает поведенческие аспекты агентов.

1.2.2. *Экспликативное программирование* (*ЭП*)

Данное программирование [27–29] ориентировано на разработку теории дескриптивных и декларативных программных формализмов, адекватных моделям структур данных, программ и средств конструирования из них программ. Для этих структур решены проблемы существования, единства и эффективности. Теоретическую основу ЭП составляет логика, конструктивная математика, информатика, композиционное программирование и классическая теории алгоритмов. Для изображения алгоритмов программ используются разные алгоритмические языки и методы программирования: функциональное, логическое, структурное, денотационное и др.

К основным принципам ЭП относятся следующие.

Принцип развития определения понятия программы в абстрактном представлении и постепенной ее конкретизации с помощью экспликаций.

Принцип прагматичности, или полезности, определения понятия программы выполняется с точки зрения понятия «проблема» и ориентирован на решение задач пользователя.

Принцип адекватности ориентирован на абстрактное построение программ и решение проблемы с учетом *информационности данных* и *апликативности*, т.е. рассмотрение программы как функции, вырабатывающей выходные данные на основе входных данных. Функция является объектом, которому сопоставляется *денотат* имени функции с помощью отношения *именования* (*номинации*).

Развитие понятия *функции* осуществляется с помощью *принципа композиционности*, сущность которого состоит в построении программы (функции) с помощью композиций из более простых программ. При этом создаются новые объекты с более сложными именами, описывающими функции и включающими номинативные (именные) выражения, языковые выражения, термы и формулы. Согласно теории Фреге, сложные имена являются дескрипциями.

Принцип дескриптивности позволяет трактовать программу как сложные дескрипции, построенные из простых программ и композиций отображения входных данных в результаты по *принципу вычислимости.*

Таким образом, процесс создания программ осуществляется с помощью цепочки понятий: данные–функция–имя, функции–композиция–дескрипция. Понятия данные–функция–композиция задают *семантический аспект* программы, а данные–имя функции–дескрипция – *синтаксический аспект*. Главным в ЭП являются семантический аспект, система композиций и номинативности (КНС), ориентированные на систематическое изучение номинативных отношений при построении данных, функций, композиций и дескрипций. КНС задают специальные языковые системы для описания разнообразных классов функций и называются *композиционно-номинативными языками функций*.

Такие системы тесно связаны с алгебрами функций и данных, они построены в семантико-синтаксическом стиле. КНС отличается от традиционных систем (моделей программ) теоретико-функциональным подходом, классами однозначных n-арных функций, номинативными отображениями и структурами данных. Они используются для построения математически простых и адекватных моделей программ параметрического типа с применением методов универсальной алгебры, математической логики и теории алгоритмов. Данные в КНС рассматриваются на трех уровнях: *абстрактном, булевом и номинативном*. Класс номинативных данных обеспечивает построение именных данных, многозначных номинативных данных или мультиименных данных, задаваемых рекурсивно.

Разработаны новые средства [26] для определения систем данных, функций и композиций номинативного типа, имена аргументов которых принадлежат некоторому множеству имен *Z*, т.е. композиция определяется на *Z-номинативных* наборах именных функций.

Номинативные данные позволяют задавать структуры данных, которым присущи неоднозначность именования компонентов типа множества, мультимножества, реляции и т.п.

Функции обладают свойством апликативности, их абстракции задают соответственно классы слабых и сильных аппликативных функций. Слабые функции позволяют задавать вычисление значений на множестве входных данных, а сильные – обеспечивают вычисление функций на заданных данных.

Композиции классифицируются уровнями данных и функций, а также типами аргументов. Экспликация композиций соответствует абстрактному рассмотрению функций как слабо аппликативных, и их уточнение строится на основе понятия *детерминанта композиции* как отображения специального типа. Класс аппликативных композиций предназначен для конструирования широкого класса программ. Доказана теорема о сходимости класса таких композиций в классе монотонных композиций.

Инструменты. Системы данных, функций и композиций реализованы в *классе манипуляционных* данных в БД, информационных системах и позволяют значительно ускорить процесс обработки запросов, заданных в SQL-подобных языках [33]. Практическая проверка теоретического аппарата формализации дедуктивных и ОО БД прошла в ряде экспериментальных проектов.

Разработана В.Н. Редько общая теория программирования, *программология*, объединяющая идеи логики, конструктивной математики и информатики, *уточняющая* понятие программы и программирования на единой концептуальной основе. Теория предоставляет общий формальный аппарат для конструирования программ. В числе наиважнейших программных понятий и принципов выделяются понятие *композиции* и *принцип композиционности*, который трактует программу как функцию, строящуюся из других функций с помощью специальных *операций*, называемых *композициями*. Принцип композиционности стал основным в ***композиционном программировании***. На основе композиционной *экспликации* (от explication – уточнение, разъяснение) понятия программирования была развита логико-математическая система композиционного построения программ, получившая впоследствии название ***экспликативного программирования*** [28]. Оно интегрирует в себе все наиболее существенные парадигмы (стили) программирования (структурное, функциональное, объектно-ориенти рованное и др.) в рамках концептуально единой экспли-

кативной платформы, основу которой составляют три основных типа объектов: собственно объекты, средства построения из одних объектов других (функции) и программологические средства применения средств построения (композиции). Прагматически ориентированную конкретизацию экспликативного программирования (в выделенным эталонным ядром из совокупности композиций) представляет собой эталонное программирование.

1.2.3. *Парадигма модульного проектирования систем*

Модульное программирования основывается на понятии *модуля*, который осуществляет преобразование множества исходных данных *X* во множество выходных данных *Y* и задается в виде отображения [30]:

$$M : X \rightarrow Y. \qquad (1.1)$$

На множества *X*, *Y* и отображение *M* накладывается ряд ограничений и дополнительных условий, позволяющих отделить модуль как самостоятельный программный объект от других классов программных объектов. Модуль обладает множеством свойств и характеристик. Эти свойства проявляются на трех основных процессах разработки системы – проектирование, разработка и выполнение.

Проектирование ПС включает следующие свойства:

– выполнение одной или более взаимосвязанных функций;

– логическую законченность функции;

– независимость одного модуля от других (внутренняя логика данного модуля не связана с логикой работы других);

– замену отдельного модуля без нарушения структуры;

– вызов других модулей и возврат данных вызвавшему модулю;

– уникальность именования модуля и др.

Процесс разработки модулей основан на следующих описаниях:

– модулей на одном из ЯП и представлении их в виде КПИ;

– характеристик модуля в паспорте;

– ограничений на размер модуля.

Процесс выполнения ПС предъявляет такие требования к свойствам модуля:

– возможность использования КПИ в различных местах ПС;

– передача данных между модулями через вызов CALL;

– изменение передаваемых данных и др.

Не все свойства одинаково важны в процессе разработки ПС. Однако для модульного проектирования характерны две тенденции – усиление *внутренних связей* в модуле и ослабление *внешних связей.* Хорошо спроектированный модуль обладает значительно более сильными внутренними связями, чем внешними. При выполнении этого условия процесс сборки мо-

дулей можно рассматривать как неделимые программные единицы с определенными свойствами без учёта их внутренней структуры. Исходя из этого, выделяются свойства модулей, наиболее важные для процесса сборки. Это логическая законченность процесса проектирования; эаменяемость отдельного модуля в ПС; возврат управления вызывающему модулю; спецификация паспорта и текста модуля в ЯП и др.

Приведенные свойства позволяют выбрать аксиоматический подход к определению модуля.

Модулем в рамках метода модульного программирования называется программный объект, характеризующийся приведенными свойствами. Это определение задает границы применения понятия *модуль*.

Определение типов связей модулей

Под *видом связи* между модулями будем понимать:

1) связь по управлению; 2) связь по данным.

1. **Связь по управлению** характеризуется наличием или отсутствием среды ЯП и механизмом вызова.

Среда ЯП определяется как совокупность программных средств окружения модулей с ЯП и включающих:

- системные средства процессов выполнения библиотечных программ связи с ОС, вычисление стандартных функций, обработки внутренних структур данных и т.д.;
- внутренние структуры данных, точки передачи и управления.

Механизм вызова через CALL в ЯП.

Связь по управлению обладает следующей функцией:

$$CP = K_1 + K_2, \tag{1.2}$$

где K_1 – коэффициент механизма вызова; K_2 – коэффициент перехода от среды ЯП вызывающего модуля к среде ЯП вызываемого.

Для стандартного механизма вызова $K_1 = 1$, для нестандартного – $K_1 = 1 + a$ ($a > 0$). Здесь a зависит от количества отличных от стандартных характеристик вызова.

К характеристикам вызова относятся: способ определения точки входа в вызываемый модуль; механизм передачи управления; формирование адреса возврата в вызывающий модуль; доступ к списку параметров; метод сохранения и восстановления регистров вызывающего модуля.

Коэффициент K_2 зависит от количества операций, необходимых для перехода от среды вызывающего модуля к среде вызываемого и наоборот. Аналитического выражения для K_2 не существует, но можно указать параметры, от которых он зависит. К ним относятся: количество библиотечных модулей, входящих в среду; количество выполняемых ими функций; количество структур данных, входящих в среду; ЯП вызывающего и вызываемого

модулей. Если вызывающий и вызываемый модули написаны на одном ЯП, то $K_2 = 0$. Для остальных случаев $K_2 > 0$.

2. **Связь по данным.** Различают регулярную и нерегулярную связи. Регулярная связь характеризуется целенаправленным обменом определенного множества данных при каждой активизации вызываемого модуля с помощью оператора вызова CALL, нерегулярная – непостоянством обмениваемых данных через посредников.

К этому типу связи относится обмен данными с глобальными именами или внешними файлами. Эта связь характеризует *сложность* ПС, описываемую следующей функцией:

$$CI = \sum_{i=1}^{n} K_i F(x_i), \qquad (1.3)$$

где K_i – весовой коэффициент для *i*-го параметра; $F(x_i)$ – функция количества элементов простых типов для параметра x_i.

Коэффициенты $K_i = 1$ – для простых переменных и $K_i > 1$ – для сложных типов данных. $F(x_i) = 1$, если x_i – простая переменная, и $F(x_i) > 1$ – для сложных типов данных. Необходимо отметить, что в (1.3) входят данные, принадлежащие к регулярной и нерегулярной информационной связи. Определение типов данных будет приведено ниже.

Интерфейсы программ и их функции

Интерфейс обозначает взаимосвязь программных объектов (модулей и программ), обеспечивающий условия их совместного функционирования. Он разрабатывается как самостоятельный программный элемент, в основе которого лежит:

– стандартизация объектов, их свойств и характеристик;

– формализация правил связи стандартизованных объектов;

– автоматизация механизмов связи объектов.

При этом имеет место несколько видов интерфейсов: межъязыковый, межмодульный, межпрограммный и пользовательский.

Межъязыковый интерфейс – это связь различных ЯП по типам содержащихся в них данных, методам их организации и способам отображения этих средств соответствующими системами программирования. В его функции входят следующее.

1. Обеспечение перехода от среды функционирования одного ЯП к среде функционирования другого. При связи разноязыковых модулей необходимо осуществить переход от среды ЯП вызывающего модуля к среде ЯП вызываемого.

2. Обеспечение передачи управления между разноязыковыми модулями при условии: если реализованы средства перехода от одной среды функцио-

нирования к другой, то передача управления между разноязыковыми модулями не отличается от передачи управления между одноязыковыми. В этом случае задача для межъязыкового интерфейса сводится к контролю за последовательностью обращения от вызывающего модуля к вызываемому.

3. Обеспечение доступа к общим данным через механизмы нерегулярной информационной связи в зависимости от места расположения информации. Основным механизмом обмена является аппарат передачи данных через параметр вызова CALL.

4. Реализация механизма передачи данных через параметры вызова предполагает преобразование данных из списка фактических параметров вызывающего модуля к представлению, согласующемуся с описанием формальных параметров вызываемого модуля. Преобразование устраняет отличия как языковых, так и реализованных системами программирования. Оно выполняется перед и после выполнения вызываемого модуля.

Межмодульный интерфейс – это обеспечение связи модулей, записанных в разных ЯП, и управление модульными структурами:

1) решение задачи сопряжения пар модулей и задачи управления модулями;
2) описание разнородных модулей в ЯП;
3) реализация модулей в системах программирования с ЯП.

1. **Отличия** в языковых средствах ЯП являются следствием неодинаковости синтаксического и семантического представления типов данных ЯП, их функциональных возможностей. К ним относятся:

- механизм конструирования новых типов данных, который отсутствует в языках Фортран, ПЛ/1, Кобол и имеется в языках Паскаль, Ада, Симула-67, Модула-2, Си, CLU;
- некоторые предопределенные типы, которые отсутствуют в некоторых ЯП (например, символьный тип в ЯП Фортран);
- формат представления разный (логический тип в языке ПЛ/1 представлен как битовая строка);
- динамические типы данных отсутствуют в Фортране и Коболе и имеются в языках Паскаль, Ада, Симула-67 и др.;
- организация внешних файлов в ЯП различна;
- дескрипторы для представления структурных типов данных имеются в некоторых ЯП и не требуются в Фортране и Коболе;
- представление некоторых структурных типов отличается в различных ЯП (массивы в Фортране располагаются по столбцам, в других ЯП – по строкам).

2. **Проблемы сопряжения** связаны с описаниями модулей, вызванными несоответствием задания формальных и фактических параметров, и состоят в следующем:

– описания типов данных, областей значений переменных, индексов массивов и т.д. задаются неоднозначно;
– с одним формальным параметром сопоставляется несколько фактических и наоборот;
– изменение порядка следования параметров.

3. **К проблемам реализации** модулей относятся:
– особенности передачи управления (наличие среды функционирования);
– различия во внутреннем представлении однородных типов данных для различных систем программирования;
– различия в структуре и организации внешней памяти для однородных файлов.

Из приведенного выше следует, что проблема передачи управления между разноязыковыми модулями носит не принципиальный характер, а является следствием реализации конкретных систем программирования с ЯП. Это подтверждается также тем, что аналогичные проблемы для ЯП, реализованных на мини- и микро-ЭВМ, практически отсутствуют или незначительны.

Управление построением и обработкой модульных структур программ как одной из задач межпрограммного интерфейса основывается на реализации следующих четырех функций.

1. Комплектование программных средств различного уровня сложности. Этот процесс выполняется несколькими процессами. На каждом приходится иметь дело с программами различного уровня сложности, готовности и отлаженности. Комплексирование (сборка) разнородных программ происходит в рамках задачи межпрограммного интерфейса.

2. Обеспечение построения различных видов программных структур (простая и динамическая структуры). Построение этих программных структур и составляет вторую задачу управления модульными структурами.

Выполнение операций над модульными структурами. Перед процессом комплексирования необходимо исследовать модульные структуры объектов для обеспечения правильности их функционирования. В частности, эти исследования состоят в выполнении операций определения доступности модулей и их именования, анализа циклов в последовательностях обращений между модулями и др. Поэтому в управлении обработкой модульных структур заключается третья задача межпрограммного интерфейса.

Обеспечение тестирования и отладки межмодульных переходов. Данная задача – комплексная, она способствует построению правильных модульных структур. При ее решении используются как средства межъязыкового интерфейса, так и средства управления модульными структурами. Средства сопряжения модулей позволяют обеспечить тестирование передава-

емых параметров, а средства управления модульными структурами – отслеживание цепочек выполнения модулей.

К задачам управления модульными структурами относятся:

1) выполнение операций над модульными структурами;
2) построение новых структур на основе модулей;
3) построение отладочной среды модульной структуры;
4) сборка модулей в агрегат с учетом решения задач 1–3.

При решении первых трех задач четвертая является тривиальной, и ее обычно выполняет специальный компонент операционной системы – редактор связей в ОС и др.

Определение модульной структуры ПС

Для представления ПС из модулей используется математический аппарат теории графов, в котором *графом G* называется пара объектов $G = (X, \Gamma)$, где X – конечное множество, называемое *множеством вершин*, а Γ – конечное подмножество прямого произведения $X \times X \times Z$, называемое *множеством дуг графа*. Из данного определения следует, что граф G фактически является мультиграфом, так как две его вершины могут быть соединены несколькими дугами. Для отличия таких дуг они нумеруются целыми положительными числами.

Определение 1.1. Модель модульной структуры ПС – это объект, описываемый тройкой $T = (G, Y, F)$, где $G = (X, \Gamma)$ – ориентированный граф, являющийся графом модульной структуры; Y – множество модулей, входящих в программный агрегат; F – функция соответствия, ставящая каждой вершине X-графа элемент множества Y.

Функция F отображает X на Y:

$$F : X \rightarrow Y. \tag{1.4}$$

В общем случае элементу из Y может соответствовать несколько вершин из X, что характерно для динамической структуры ПС.

Определение 1.2. Модульной структурой ПС называется пара $S = (T, \chi)$, где T – модель модульной структуры ПС; χ – характеристическая функция, определенная на множестве вершин X графа модулей G.

Значение функции χ определяется следующим образом:

$\chi(x) = 1$, если модуль, соответствующий вершине $x \in X$, включен в состав ПС программного агрегата;

$\chi(x) = 0$, если модуль, соответствующий вершине $x \in X$, не включен в состав ПС и к ней имеются обращения из других модулей.

Определение 1.3. Две модели модульных структур $T_1 = (G_l, Y_1, F_1)$ и $T_2 = (G_2, Y_2, F_2)$ тождественны, если $G_1 = G_2$, $Y_1 = Y_2$, $F_1 = F_2$. Модель T_1 изоморфна модели T_2, если $G_1 = G_2$ между множествами Y_1 и Y_2 существует изоморфизм φ, а для любого $x \in X \Rightarrow F_2(x) = \varphi(f_1(x))$.

Определение 1.4. Две модульные структуры $S_1 = (T_1, \chi_1)$ и $S_2 = (T_2, \chi_2)$ тождественны, если $T_1 = T_2$ и $\chi_1 = \chi_2$, модульные структуры S_1 и S_2 называются *изоморфными*, если T_1 изоморфна T_2 и $\chi_1 = \chi_2$.

Понятие изоморфизма модульных структур и их моделей необходимо для введения уровня абстракции, на котором определяются операции над модулями. Для изоморфных объектов операции будут интерпретироваться одинаково без ориентации на конкретный состав модулей. Данные операции определяются над парами (G, χ).

Введенные выше определения описывают модульные структуры общего вида. Свойства модульных структур состоят в следующем:

– граф модульной структуры G имеет один или несколько компонентов связности, каждая из которых представляет ациклический граф;

– в каждом компоненте графа выделена единственная вершина, которая называется *корневой* и характеризуется тем, что не существует входящих в нее дуг и соответствующий ей модуль выполняется первым (для связности);

– циклы допускаются только для случая, когда соответствующий некоторой вершине модуль имеет рекурсивное обращение к самому себе. Обычно такая возможность реализуется компилятором с соответствующего ЯП и данный тип связи не рассматривается межмодульным интерфейсом;

– пустой граф G_0 соответствует пустой модульной структуре.

В дальнейшем под терминами *модульная структура, граф модульной структуры* понимаются объекты, удовлетворяющие указанным выше условиям. Типичный пример графа приведен на рис. 1.1.

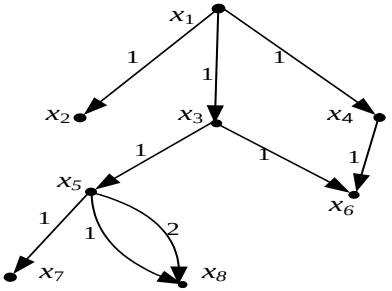

Рис. 1.1. Пример графа модульной структуры программы

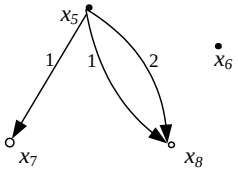

Рис. 1.2. Граф структуры модулей x_5 и x_6

Вершины $x_1, x_2, ..., x_8$ составляют множество X. Все дуги пронумерованы. Из модуля с вершиной x_5 имеются два обращения к модулю с вершиной x_8. Множество дуг графа имеет вид $\Gamma = \{(x_1, x_2, 1), (x_1, x_3, 1), ..., (x_5, x_8, 1), (x_5, x_8, 2)\}$. В дальнейшем этот граф будет использоваться для иллюстрации

операций над модульными структурами. Так, на рис. 1.2. показана структура графа, соответствующая модулям x_5 и x_6, а на рис. 1.3. – модульные структуры для трех видов сегментов.

1.2.3.1. Матричное представление графов из модулей

Для определения основных операций над модульными структурами используем матричное представление их графов. Матричные представления часто используются в различных работах. Например, матрицы вызовов и матрицы достижимости эквивалентны матрицам смежности ориентированных графов.

В настоящей работе в качестве матричного представления используется матрица вызовов. Элемент матрицы m_{ij} определяет количество обращений (операторов вызова) из модуля, соответствующего индексу i, к модулю, соответствующему индексу j. Кроме матрицы вызовов для дальнейшего анализа используется характеристический вектор, для каждого компонента i которого $V_i = \chi(x_i)$. Для графа модульной структуры, приведенной на рис. 1.1, характеристический вектор и матрица вызовов имеют вид:

$$V = \begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{pmatrix}, \quad M = \begin{pmatrix} 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 2 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}. \qquad (1.5)$$

Проведем анализ матриц вызовов и характеристических векторов для графов модульных структур, соответствующих различным типам программных агрегатов. Для модулей с графами, представленными на рис. 1.6, векторы и матрицы имеют следующий вид:

$$V_5^m = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}, \quad M_5^m = \begin{pmatrix} 0 & 1 & 2 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix};$$

$$V_6^m = (1), \quad M_6^m = (0). \qquad (1.6)$$

Только один элемент характеристического вектора равен единице и только в одной строке матрицы находятся ненулевые элементы. На рис. 1.2 векторы и матрицы записываются в таком виде:

$$V_3^s = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}, \quad M_3^s = \begin{pmatrix} 0 & 1 & 1 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}; \quad V_1^s = \begin{pmatrix} 1 \\ 1 \\ 0 \\ 0 \end{pmatrix}, \quad M_1^s = \begin{pmatrix} 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix};$$

$$V_5^s = \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix}, \quad M_5^s = \begin{pmatrix} 0 & 1 & 2 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}. \qquad (1.7)$$

Для программы с графом, представленным на рис. 1.1, характеристический вектор и матрица вызовов совпадают с V и M соответственно и определяются в (1.4). Все элементы V равны единице.

В комплексе программ характеристический вектор и матрица вызовов имеют следующий вид:

$$V^c = \begin{pmatrix} V_1^p \\ V_2^p \\ \dots \\ V_n^p \end{pmatrix}, \quad M^c = \begin{pmatrix} M_1^p & 0 & \dots & 0 \\ 0 & M_2^p & \dots & 0 \\ \dots & \dots & \dots & \dots \\ 0 & 0 & \dots & M_n^p \end{pmatrix}. \qquad (1.8)$$

Здесь V_i^p и M_i^p ($i = \overline{1, n}$) обозначают характеристический вектор и матрицу вызовов для графа i-й программы, входящей в комплекс.

В дальнейшем матричное представление используется для анализа операций над модульными структурами.

Отношение достижимости для графов модулей

Пусть $G = (X, \Gamma)$ – граф модульной структуры, x_i, x_j – вершины, принадлежащие X. Если в графе G существует ориентированная цепь от x_i к x_j, то вершина x_j – достижима из вершины x_i. Справедливо следующее утверждение: если вершина x_j достижима из x_i, а x_l – из x_j, то x_l достижима из x_i. Доказательство этого факта очевидно. Рассмотрим бинарное отношение на множестве X, которое определяет достижимость между вершинами графа. Введем обозначение $x_i \to x_j$ для достижимости вершины x_j из x_i. Отношение транзитивно. Обозначим через $D(x_i)$ множество вершин графа G, достижимых из x_i. Тогда равенство

$$\overline{x_i} = \{x_i\} \cup D(x_i) \qquad (1.9)$$

определяет транзитивное замыкание x_i по отношению достижимости.

Докажем следующую теорему.

Теорема 1.1. *Для выбранного модуля связности графа модульной структуры любая вершина достижима из корневой, соответствующей данному модулю, т.е. выполняется равенство* (x_1 *– корневая вершина*)

$$\overline{x_1} = \{x_1\} \cup D(x_1) = X. \qquad (1.10)$$

Доказательство. Предположим, вершина x_i ($x_i \in X$) недостижима из x_1. Тогда $x_i \notin \overline{x_1}$, и множество $X' = X \setminus \overline{x_1}$ непусто. Поскольку выбранный компонент графа связанный, то существуют вершина $x_j \in \overline{x_1}$ и цепь $H(x_i, x_j)$, ведущая от x_i к x_j. Исходя из ацикличности графа G, в X' должна существовать простая цепь $H(x_l, x_j)$, где в вершину x_l не входят дуги (данная цепь может быть пустой, если X' состоит только из x_i). Рассмотрим цепь $H(x_l, x_j) = H(x_l, x_i) \cup H(x_i, x_j)$. Это означает, что модуль x_j достижим из вершин x_1 и x_i и обе вершины не содержат входящих дуг. А это противоречит определению графа модульной структуры с единственной корневой вершиной. Теорема доказана.

Результаты данной теоремы важны для обоснования требования отсутствия ориентированных циклов в графе модульной структуры относительно понятия достижимости. Исходя из графа на рис. 1.4, отмечаем, что граф содержит ориентированный цикл и модули, соответствующие вершинам x_4, x_5, x_6, никогда выполняться не будут. Таким образом, результаты теоремы 1.1 усиливают требование отсутствия ориентированных циклов в графе модулей.

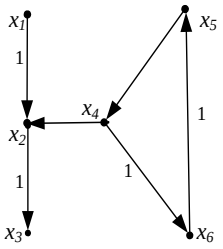

Рис. 1.4. Граф
с ориентированным циклом

Для анализа матричного представления отношения достижимости графа модульной структуры рассматривается граф матрицы достижимости, приведенной на рис. 1.1.

$$\mathbf{A} = \begin{pmatrix} x_1 & x_2 & x_3 & x_4 & x_5 & x_6 & x_7 & x_8 \\ 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix} \qquad (1.11)$$

Коэффициент $a_{ij} = 1$, если модуль, соответствующий индексу l, достижим из модуля, соответствующего индексу i. Следующие результаты основаны на известной теореме из теории графов.

Теорема 1.2. *Коэффициент m_{ij} l-й степени матрицы смежности M определяет количество различных маршрутов, содержащих l дуг и с связывающих вершину x_i с вершиной x_j-ориентированного графа.*

Доказательство этой теоремы не приводим. Рассмотрим следующие три следствия из этой теоремы.

Следствие 1.1. Матрица $\overline{M} = \sum_{l=1}^{n} M^i$, где M – матрица смежности ориентированного графа с n вершинами совпадает с точностью до числовых значений коэффициентов с матрицей достижимости A.

Доказательство. В ориентированном графе, содержащем n вершин, максимальная длина пути без повторяющихся дуг не может превышать n. Поэтому последовательность степеней матрицы смежности M^i, где i = 1, 2, …, n определяет количество всех возможных путей в графе с количеством дуг $\le n$. Пусть коэффициент $\overline{m_{ij}}$ матрицы M отличен от нуля. Это означает, что существует степень матрицы M^i, у которой соответствующий коэффициент m_{ij} также отличен от нуля. Следовательно, существует путь, идущий от вершины x_i к x_j, т.е. вершина x_j достижима из x_i. Данное следствие определяет связь матрицы вызовов графа модульной структуры, совпадающей с матрицей смежности M, с матрицей достижимости A и определяет алгоритм построения последней.

Следствие 1.2. Пусть для некоторого i в последовательности степеней матрицы смежности M^i существует коэффициент $m_{ii} > 0$. Тогда в исходном графе существует ориентированный цикл.

Доказательство. Пусть $m_{ii} > 0$ для некоторого l. Следовательно, x_l достижима из x_i, т.е. существует цикл. Согласно теореме 4.2 данный цикл имеет l дуг (в общем случае повторяющихся).

Следствие 1.3. Пусть n-я степень матрицы смежности M^n ациклического графа совпадает с нулевой матрицей (все коэффициенты равны нулю).

Доказательство. Если граф ациклический, то в нем максимально простой путь не может иметь больше чем $n - 1$ дуг. Если в M^n имеется коэффициент, отличный от нуля, то должен существовать путь, состоящий из n дуг. А этим путем может быть только ориентированный цикл. Следовательно, все коэффициенты M^n для ациклического графа равны нулю. Данное следствие предоставляет необходимое и достаточное условие отсутствия циклов в графе модульной структуры.

Для ациклических графов отношение достижимости эквивалентно частичному строгому порядку. Транзитивность отношения достижимости рассмотрена выше. Антисимметричность следует из отсутствия ориенти-

рованных циклов: если вершина x_j достижима из x^l, то обратное неверно. Введем обозначение $x_i > x_j$, если вершина x_j достижима из вершины x_i.

Пусть $G = (X, \Gamma)$ – ациклический граф, соответствующий некоторой модульной структуре. Рассмотрим убывающую цепь элементов частично упорядоченного множества X:

$$x_{i1} > x_{i2} > \ldots > x_{in} \ldots,$$

где через «>» обозначено отношение достижимости. Поскольку X конечно, то цепь обрывается $x_{i1} > x_{i2} > \ldots > x_{in}$. Вершина x_{in} не имеет исходящих дуг, т.е. элемент x_{in} минимальный (ему соответствует модуль, который не содержит обращения к другим модулям). Максимальный элемент во множестве X – корневая вершина.

Операции построения модульных структур

Выше было отмечено, что операции над модульными структурами выполняются на уровне абстракции, определяемом понятием *изоморфизма модульных структур.* Над изоморфными структурами операции выполняются одинаково, поэтому преобразование модульных структур будет рассматриваться как изменение их графов и характеристических функций, т.е. $S = (G, \chi)$.

Пусть $S_1 = (G_1, \chi_1)$ и $S_2 = (G_2, \chi_2)$ – две модульные структуры с графами $G_1 = (X_1, \Gamma_1)$ и $G_2 = (X_2, \Gamma_2)$ соответственно. Введем следующие обозначения:

$D(x)$ – множество вершин, достижимых из вершины x;

$D^*(x)$ – множество вершин, из которых достижима вершина x.

Для одинаковых вершин, входящих в графы G_1 и G_2, будут использоваться одинаковые обозначения.

Рассмотрим основные операции над модульными структурами. Операция *объединения*

$$S = S_1 \bigcup S_2 \qquad (1.12)$$

предназначена для формирования модульной структуры комплекса и формально определяется следующим образом (S_1 и S_2 – любые модульные структуры, удовлетворяющие определению в п. 1.1):

$$G = G_1 \oplus G_2, \quad X = X_1 \oplus X_2, \quad \Gamma_1 \oplus \Gamma_2, \qquad (1.13)$$

где символ $\oplus$ обозначает прямую сумму, и

$$\chi(x) = \chi_1(x), \text{ если } \chi X_1,$$
$$\chi(x) = \chi_2(x), \text{ если } \chi X_2.$$

Одинаковые вершины, входящие в G_1 и G_2, операцией объединения модульных структур рассматриваются как разные объекты. Поэтому характе-

ристический вектор и матрица вызовов для модульной структуры S определяются так:

$$V_{1,2} = \begin{pmatrix} V_1 \\ V_2 \end{pmatrix}, \quad M_{1,2} = \begin{pmatrix} M_1 & 0 \\ 0 & M_2 \end{pmatrix}, \tag{1.14}$$

где $V_{1,2}$ и $M_{1,2}$ – характеристические векторы и матрицы вызовов для модульных структур S_1 и S_2 соответственно. Операция объединения ассоциативна, но не коммутативна – порядок следования операндов определяет порядок выполнения компонентов комплекса. Необходимо отметить, что если операнды S_1 и S_2 удовлетворяют условиям определения модульных структур, то результат S будет удовлетворять тем же требованиям.

Операция объединения модульных структур увеличивает число компонентов связности соответствующего графа. Кроме того, графы структур, соответствующие операндам, сами могут иметь несколько компонентов связности. Для остальных операций графы модульных структур операндов и результата имеют единственный компонент связности.

Рассмотрим операцию *соединения*. Через x_i и x_j обозначим корневые вершины для графов G_1 и G_2 модульных структур S_1 и S_2 соответственно. Если данные структуры удовлетворяют условиям:

множество $X' = X_1 \bigcap X_2$ непустое;

вершина $x_j \in X'$ и $\chi(x_j) = 0$;

$D^*(x) \bigcap D(x) = 0$ для каждого $x \in X'$, где $D^*(x) \in X_1$ и $D(x) \in X_2$, то операция соединения, обозначаемая

$$S = S_1 + S_2, \tag{1.15}$$

определяется следующим образом:

$$G = G_1 \bigcup G_2, \quad X = X_1 \bigcup X_2, \quad \Gamma = \Gamma_1 \bigcup \Gamma_2, \tag{1.16}$$

и для характеристической функции χ выполняется:

$$\chi(x) = \chi_1(x), \text{ если } x \in X_1 \setminus X';$$

$$X(x) = \max(\chi_1(x), \chi_2(x)), \text{ если } x \in X',$$

$$\chi(x) = \chi_2(x), \text{ если } x \in X_2 \setminus X'.$$

Первое условие означает, что в графах G_1 и G_2 имеются общие вершины. Согласно второму условию корневая вершина G_2 принадлежит общей части и для S_1 объект, соответствующий x_j, еще не включен в модульную структуру. Третье условие запрещает существование циклов в графе результата. Действительно, если существует $x_n \in D^*(x) \bigcap D(x)$, то $x_n > x$ и

$x > x_n$, что означает существование цикла. Так как не все S_1 и S_2 удовлетворяют приведенным выше условиям, то операция частичная.

Определим принадлежность результата операции соединения к классу рассматриваемых модульных структур. Поскольку X' непусто, то граф G имеет один компонент связности. Корневой вершиной графа G является x_i. Сам граф G не имеет ориентированных циклов, т.е. ацикличен. Таким образом, S принадлежит к классу рассматриваемых модульных структур.

Операция соединения не коммутативная и в общем случае не ассоциативна. Чтобы показать последний факт, рассмотрим результат $S = (S_1 + S_2) + S_3$, где корневые вершины графов G_2 и G_3 входят в состав вершин графа G_1 и $X_2 \bigcap X_3 \neq 0$. Тогда результат $S_2 + S_3$ не определен.

Операция проекции. Пусть $S_1 = (G_1, \chi_1)$ – модульная структура и $x_i \in X_1$. Операция проекции модульной структуры на вершину графа S_1, обозначаемая как $S = P_{rxi}(S_1)$, определяется следующим образом:

$$G\left(X,\Gamma\right),\ \ X=\overline{x}_i,\ \Gamma=\left\{\left(x_i,x_j,K\right)\mid x_i,x_j\in X\right\}, \tag{1.17}$$

и для характеристической функции

$$\chi\left(x\right)=\ \chi_1\left(x\right),\ \text{если}\ x\in X.$$

Операция проекции определяет из модульной структуры S_1 подструктуру S. Проверим принадлежность S классу рассматриваемых модульных структур. Если граф модульной структуры S_1 связан и ацикличен, то теми же свойствами будет обладать и граф S. Существует единственная корневая вершина x_i в графе G. Таким образом, модульная структура S принадлежит рассматриваемому классу.

Операция разности для модульных структур определяется следующим образом. Пусть $S_1 = (G_1, \chi_1)$ – модульная структура и $x_i \in X_1$. Операция разности выполняется между модульной структурой и ее проекцией на вершину x_i соответствующего графа (x_i не является корневой вершиной графа G_1). Формально операция разности модульной структуры

$$S\ =\ S_1-P_{r_{xi}}\left(S_1\right), \tag{1.18}$$

определяется следующим образом:

$$G\ =(X,\Gamma),\ \ X=(X_1\setminus\overline{x_i})\bigcup X',\tag{1.19}$$

$$\Gamma=\left(x_i,x_j,K\right)\mid x_i,x_j\in X,$$

где множество X' состоит из таких элементов, для которых

$$X'=\{x'_j\mid(x_l\in X_1\setminus\ x_i)\&(x'_j\in\overline{x_i})\&\left(x_l,x'_j,K\right)\in\Gamma\}. \tag{1.20}$$

Характеристическая функция χ определяется так:

$\chi(x) = \chi_1(x)$, если $x \in X_1 \setminus \overline{x_i}$;

$\chi(x) = 0$, если $x \in X'$.

Во множество X включаются вершины, которые не вошли во множество $\overline{x_i}$, и те из вершин $\overline{x_i}$, в которые входят дуги из вершины $X_1 \setminus \overline{x_i}$ (множество X'). Характеристическая функция для элементов $x' \in X'$ равна нулю. Операция разности модульных структур определена так, чтобы быть обратной к операции соединения, т.е. чтобы выполнялось равенство

$$S - P_{r_{xi}}(S) + (S) = S. \qquad (1.21)$$

Проверим принадлежность S, определяемой в (1.15), к классу рассматриваемых модульных структур. Если граф G, связан и цикличен, то этими же свойствами будет обладать граф G_1. Корневая вершина G совпадает с корневой вершиной G_1. Таким образом, S удовлетворяет условиям определения модульной структуры, приведенным в п. 1.1.

Пусть S^* обозначает множество модульных структур, заданное на прямом произведении $G^* \; X \; \chi^*$, где G^* и χ^* – соответственно множество графов и характеристических функций. Обозначим через Ω множество введенных операций над модульными структурами и предикаты C и R, рассмотренные выше:

$$\Omega = \{+, -, P_r, C, R\}. \qquad (1.22)$$

Этим мы определяем частичную алгебраическую систему $U = (S^*, \Omega)$ над множеством модульных структур. Согласно определению типов программных агрегатов и операций над модульными структурами следующего вида выделяются:

– объединение модулей в программы и комплексы;

– соединение модулей и сегментов для получения программы;

– проекция программ и сегментов для получения модуля;

– разность программ и сегментов для получения модуля.

Процесс построения модульных структур

Рассмотрим основные задачи построения программных агрегатов и алгоритмы их решения. Они взяты из практики комплексирования программных средств и этим во многом определилась реализация алгоритмов решения данных задач.

Задача 1. Дано множество модулей, входящих в состав программы, и имя главного (корневого) модуля. Построить модульную структуру программы.

Задача 2. Дано множество модулей, входящих в состав программы. Имя главного модуля неизвестно. Построить структуру из модулей.

Задача 3. Дано множество модулей, реализующих некоторые функции предметной области, и имя главного модуля для одной из программ. Построить модульную структуру программы.

Задача 4. Дано множество модулей реализации функции предметной области и последовательность имен главных модулей нескольких программ. Построить модульную структуру для комплекса программ.

Задача 5. Заменить в модульной структуре один или несколько модулей новыми.

Задача 6. Выделить из модульной структуры объекты и включить их в другую структуру. В этой задаче под объектами понимается любая часть модульной структуры, т.е. любой подграф ее графа.

При решении данных задач будут использоваться введенные операции над модульными структурами. Для практического применения используется матричное представление объектов, а в качестве операций рассматриваются преобразования соответствующих матриц. Так, множества, характеристические функции представляются векторами, графы – матрицами вызовов и т.д. Аналогичные представления применяются и для объектов, используемых в алгоритмах решения задач. Технические аспекты реализации матричных представлений объектов и операций принципиальных трудностей не представляют и в данной работе не рассматриваются.

Рассмотрим решение каждой из поставленных задач, используя операции над модульными структурами.

1. Пусть X^p – множество вершин графа, соответствующее множеству модулей программы. Упорядочим его так, чтобы для каждых x_i, $x_j \in X^p$ из условия, что модуль, соответствующий x_i, вызывает модуль, соответствующий x_j, следует, что $j > i$. Если X^p таким образом не удается упорядочить, то в графе модульной структуры возникнут циклы, что противоречит необходимости ацикличности графа.

Обозначим вершину, соответствующую главному модулю, через x_1. Основной метод решения данной задачи состоит в постепенном наращивании модульной структуры «сверху вниз». С каждым модулем, соответствующим вершине x_i, связана модульная структура

$$S_i^m = (G_i^m, \chi_i^m), \text{ где } G_i^m = (\chi_i^m, \Gamma_i^m).$$

Множество χ_i^m включает x_i и вершины, соответствующие модулям, к которым имеются обращения из данного модуля. Множество Γ_i^m соответствует множеству вызовов из модуля, соответствующего x_i. Характеристическая функция для x_i равна единице и нулю – для остальных вершин. Ес-

ли из модуля, соответствующего x_i, нет обращения к другим модулям, то $X_i^m = \{x_i\}$, $\Gamma_i^m = 0$, $\chi(x_i) = 1$. Пусть Y – множество вершин графа, для которых $\chi = 0$.

Шаг 1. Вводим начальные значения: $S = S_i^m$; $Y := X_1^m \setminus \{x_1\}$.

Шаг 2. Если $C(S) = 1$, то перейти на шаг 6.

Шаг 3. Выберем первый элемент $x_i \in Y$. Множество Y конечно.

Шаг 4. $S := S + S_i^m$, $Y := (Y \setminus \{x_i\}) \bigcup (x_i^m \setminus \{x_i\})$. На данном шаге происходит наращивание модульной структуры и изменение множества Y.

Шаг 5. Переход на шаг 2.

Шаг 6. Выход.

Докажем корректность и сходимость данного алгоритма. На шаге 4 выполняется операция соединения структур. Данная операция корректна, так как ее условия выполняются:

$$x_i \in X \bigcap X_i^m,\ \chi(x_i) = 0,\ X_i^m(x_i) = 1,$$

циклы отсутствуют согласно упорядоченности множества X^p.

Рассмотрим изменение множества Y на шаге 4. Согласно результатам п. 1.1 в графе модульной структуры обязательно существуют вершины, из которых исходят дуги. Исходя из этого факта и конечности множества X^p, следует, что существует x_i такое, что $Y \setminus \{x_i\} = 0$ и $\chi_i^m \setminus \{x_i\} = 0$. В результате условие шага 2 истинно и осуществляется переход на шаг 6 – завершение работы алгоритма.

Необходимо отметить, что данный алгоритм имеет практическое воплощение в процессе комплексирования, основанном на применении специального компонента операционной системы – редактора связей, построителя задач, комплексатора и т.д.

2. Для решения второй задачи достаточно упорядочить множество X^p, как и для условия первой. Тогда первый элемент X^p соответствует главному модулю, и можно воспользоваться алгоритмом решения первой задачи.

3. Для решения третьей задачи проведем аналогичное упорядочение модулей предметной области. Единственное условие состоит в том, чтобы $x_1 \in X$ соответствовал главный модуль программы. Тогда можно воспользоваться алгоритмом решения первой задачи.

4. Пусть X^c – множество вершин, соответствующих модулям реализации функции предметной области, и $x' = \{x_1, x_2, ..., x_n\}$ – множество вершин, соответствующих главным модулям программ ($X' \subset X^c$). Алгоритм решения следующий.

Шаг 1. Начальное значение: $i := 1$, $S := 0$.

Шаг 2. Выбрать $x_i \in X'$ и построить модульную структуру с главным модулем, соответствующим корневой вершине x_i. Построение выполняется согласно алгоритму третьей задачи. Пусть S_i^p – соответствующая модульная структура.

Шаг 3. $S := S \bigcup S_i^p$. На данном шаге используется операция объединения модульных структур.

Шаг 4. $i := i + 1$. Если $i \le n$, то переход на шаг 2.

Шаг 5. Выход.

Корректность и сходимость данного алгоритма очевидны и не требуют доказательства.

5. Пусть $S = (G, \chi)$ – модульная структура с графом $G = (X, \Gamma)$. Необходимо заменить модуль программы, соответствующий вершине $x_i \in X$, новым модулем x'_i. Сложность данной задачи состоит в том, что с заменой модуля может измениться модульная структура S, так как x_i в общем случае является корневой вершиной для некоторого графа. Данный алгоритм будет иметь следующий вид.

Шаг 1. Построить модульную структуру S'_i для корневой вершины x'_i согласно алгоритмам задач 1–3 (предполагается, что множество модулей для данной структуры известно до ее построения).

Шаг 2. $S := S - P_{rx_i}(S)$. Исключается соответствующая часть модульной структуры S.

Шаг 3. Переобозначим x'_i через x_i, а S'_i – через S_i. Выполним операцию: $S = S + S_i$.

Шаг 4. В результате предыдущей операции в графе G модульной структуры могут существовать вершины с $\chi = 0$, т.е. $C(S) = 0$. Тогда необходимо применить алгоритм первой задачи, полагая в качестве множества Y множество таких вершин. Выполнение операции соединения на шаге 3 должно удовлетворять необходимым условиям.

6. Данная задача подобна задаче 5. Пусть $S_1 = (G_1, \chi_1)$ – исходная модульная структура с графом $G_1 = (X_1, \Gamma_1)$ и $x_i \in X_1$. Граф $G_2 = (X_2, \Gamma_2)$ модульной структуры $S_2 = (G_2, \chi_2)$ имеет вершину x_i с $\chi(x_i) = 0$. Тогда алгоритм решения данной задачи следующий.

Шаг 1. $S_i = P_{rxi}(S_1)$.

Шаг 2. $S := S_2 + S_i$. Операция соединения модульных структур должна удовлетворять необходимым условиям.

Шаг 3. Если $C(S) = 0$, то выполнять действия, аналогичные шагу 4 пятой задачи.

Как видно из приведенных выше алгоритмов, для решения задач 1–6 использовались все четыре операции над модульными структурами.

Проверка правильности построения модульной структуры

Существует два способа проверки. Первый основан на анализе результатов процесса редактирования связей (сборки модулей), выполняемого специальной программой операционной системы. Данный способ позволяет выявить грубые ошибки – отсутствие модулей в модульной структуре, к которым есть обращения. Эти ошибки – следствие реального отсутствия модулей или неверного имени в операторе вызова LINK.

Второй способ основан на анализе самой модульной структуры, который заключается в следующем:

1. Визуальный анализ графа модульной структуры. Проверка правильности построения непосредственно осуществляется самим разработчиком ПС. Отображение графа модульной структуры на экране терминала или вывод его на печать.

2. Анализ матриц, описывающих модульные структуры. Для проверки правильности построения модульных структур используются матрицы вызовов и достижимости. Результат проведенного анализа – установка существования циклов, числа маршрутов и достижимости между каждой парой модулей.

Таким образом, создание модульной структуры обеспечено теоретически с помощью матрицы достижимости. Аналогично, эта теория применима и для отдельных компонентов.

1.2.4. *Парадигма сборочного программирования*

Сборка – это способ соединения разноязычных объектов в ЯП, который базируется на теории спецификации и отображении типов и структур данных ЯП путем преобразования передаваемых между модулями разнородных данных [28–30]. Сборочное программирование можно толковать как синтез программы по спецификации задачи в условиях, когда отдельные элементы уже отработаны и запрограммированы. При этом синтез конкретной программы из модулей сводится к извлечению из условия задачи схемы сборки модулей, а эта работа существенно более простая, и ее легче контролировать.

Для некоторых классов задач схема сборки может извлекаться из условия задачи по формальным правилам и в результате процесса сборки программы из готовых элементов, полученных на разных методах программирования. В основе сборочного программирования лежит метод сборки, интерфейс и КПИ.

Главным объектом сборки – компоненты повторного использования (КПИ). По своей сущности, сборка из КПИ соответствует конвейерной сборке (как говорил В.М. Глушков), подобно сборки автомобилей из гото-

вых комплектующих и стыковочных деталей. В нем роль комплектующих «деталей» выполняют КПИ разной степени сложности, а роль стыковки – интерфейсы. Процесс сборки любых изделий характеризуется не только готовыми комплектующими «деталями и узлами», а и схемой сборки, задающей связи отдельных компонентов и правилами взаимодействия между собой разных объектов ТЛ на сборочном конвейере.

Объект сборки обладает свойствами (наследование, полиморфизм и инкапсуляция), включает данные и операции (методы) для установления связей между разными объектами. Объекты сборки могут обладать общими свойствами и методами, образовывать классы и быть экземплярами класса. Все объекты должны иметь паспорта, которые содержат данные, необходимые для их соединения в более сложные структуры.

Операции сборки выполняются в соответствии с паспортами объектов и правилами соединения между собой двух разноязыковых объектов. Информация в паспортах должна быть систематизирована и выделена в отдельные группы данных и их типов. Правила соединения зависят от совместимости объединяемых объектов, которые содержат описание функций, необходимых для согласования разных параметров, представленных в паспортах.

Процесс сборки объектов может проводиться ручным или автоматизированным способами. Ручной способ нецелесообразен, поскольку сборка готовых КПИ представляет собой большой объем действий, которые носят рутинный характер. Автоматизированный сборщик осуществляет сборку с помощью стандартных правил соединения разнородных объектов.

Средства, которые поддерживают процесс сборки, называют инструментальным средствами: комплексирование КПИ; генерация интерфейсных посредников; оценки показателей качества и др. Необходимые условия данного программирования:

- большое количество разнородных КПИ в библиотеках;
- паспортизация объектов сборки;
- правила сопряжения объектов с помощью средств автоматизации процесса сборки;
- принципы реализации связей между объектами сборки.

Метод сборки разноязыковых модулей в новые программные структуры включает в себя математические формализмы определения связей (по данным и по управлению) между объектами сборки и генерации интерфейсных модулей для каждой пары объединяемых модулей. Связь модулей задается оператором вызова CALL, в котором представлены фактические параметры вызываемому модулю.

Сущность задачи сборки пары разноязыковых модулей состоит в определении взаимно однозначного соответствия между задаваемым

множеством фактических параметров $V = \{v_1, v_2, ..., v_к\}$ вызывающего модуля и соответствующим множеством формальных параметров $F = \{f_1, f_2, ..., f_к\}$ вызываемого модуля, а также в отображении типов данных одних параметров в другие. Если отображение не удается выполнить, то задача автоматизированной связи данной пары модулей считается неразрешимой.

Преобразование типов данных осуществляется путем построения алгебраических систем, содержащих для каждого типа множество значений и операций над ними. Каждой операции преобразования типов данных соответствует изоморфное отображение одной алгебраической системы в другую.

Алгебраические системы построены в классе простых и сложных типов данных ЯП. Преобразования между типами массивов и записей сводятся к определению изоморфизма между основными множествами соответствующих алгебраических систем с помощью операций изменения уровня структурирования данных – селектора и конструирования. Для массива операция селектора сводится к отображению множества индексов на множество значений элементов массива. Аналогично такая операция определяется для записи как отображение между селекторами компонентов и самими компонентами.

Формально преобразование P неэквивалентных типов данных в ЯП выполняется следующими этапами.

Этап 1. Построение операций преобразования типов данных $T = \{T_\alpha{}^t\}$ для множества языков программирования $L = \{l_\alpha\}_{\alpha=1, n}$.

Этап 2. Построение отображения простых типов данных для каждой пары взаимодействующих компонентов в $l_{\alpha1}$ и $l_{\alpha2}$ и применение операций селектора S и конструктора C для отображения сложных структур данных в этих языках.

Формализованное преобразование типов данных осуществляется с помощью алгебраических систем для каждого типа данных $T_\alpha{}^t$: $G_\alpha^t = < X^t{}_\alpha, \Omega_\alpha^t >$, где t – тип данных; $X_\alpha{}^t$ – множество значений, которые могут принимать переменные этого типа; $\Omega_\alpha{}^t$ – множество операций над типами данных.

Простым и сложным типам данных современных ЯП соответствуют классы алгебраических систем:

$$\Sigma_1 = \{G_\alpha^b, G_\alpha^c, G_\alpha^i, G_\alpha^r\}, \quad \Sigma_2 = \{G_\alpha^a, G_\alpha^z, G_\alpha^u, G_\alpha^e\}. \qquad (1.23)$$

Каждый элемент класса простых и сложных типов данных определяется на множестве их значений и операций над ними: $G_\alpha^t = < X^t{}_\alpha, \Omega_\alpha^t >$, где $t = b, c, i, r, a, z, u, e$.

Операциям преобразования каждого t типа данных соответствует изоморфное отображение двух алгебраических систем с совместимыми типами данных двух разных ЯП. В классе систем Σ_1 и Σ_2 преобразование типов данных $t \to q$ для пары языков l_t и l_q обладает такими свойствами отображений:

1) $G_\alpha{}^t$ и $G_\beta{}^q$ – изоморфны (q определено на том множестве, что и t);

2) между $X_\alpha{}^t$ и $X_\beta{}^q$ существует изоморфизм, для которых множества $\Omega_\alpha{}^t$ и $\Omega_\beta{}^q$ разные. Если $\Omega = \Omega_\alpha{}^t \cap \Omega_\beta{}^q$ не пусто, то рассматриваем изоморфизм между $G_\alpha{}^{t'} = <X_\alpha{}^t, \Omega>$ и $G_\beta{}^{q'} = <X_\beta{}^q, \Omega>$. Такое преобразование сводится к случаю 1).

Между множествами $X_\alpha{}^t$ и $X_\beta{}^q$ может не существовать изоморфного соответствия. В этом случае необходимо построить такое отображение между $X_\alpha{}^t$ и $X_\beta{}^q$, чтобы оно было изоморфным. Если такое отображение существует (в каждом конкретном случае оно может быть разным), то имеем условие случая 1) с соответствующими изменениями в определении алгебраических систем;

3) мощности алгебраических систем должны быть равны $|G_\alpha{}^t| = |G_\beta{}^q|$.

Любое отображение 1), 2) сохраняет линейный порядок, так как алгебраические системы линейно упорядочены.

Лемма 1. Для любого изоморфного отображения φ между алгебраическими системами $G_\alpha{}^t$ и $G_\beta{}^q$ выполняются равенства $\varphi(X_\alpha{}^t . _{\min}) = X_\beta{}^q . _{\min}$, $\varphi(X_\alpha{}^t . _{\max}) = X_\beta{}^q{}_{\max}$.

Доказательство леммы тривиальное и простое. При условии, когда одно или два вышеприведенных равенства не выполняются, тогда для основных множеств алгебраических систем изменяется линейный порядок, что противоречит определению. Более подробно см. [6, 9, 10].

Литература к теме 1

1. *Ляпунов А.А.* О логических схемах программ // Проблемы кибернетики. – Вып. 1. – М.: Физматгиз, 1958.
2. *Янов Ю.И.* О логических схемах алгоритмов // Проблемы кибернетики. – Вып. 1. – М.: Физматгиз, 1958.
3. *Ершов А.П.* Введение в теоретическое программирование. – М., 1977.
4. *Ющенко Е.Л.* Адресный язык. – Киев: Знание, 1962. – 31 с.
5. Глушков В.М., Цейтлин Г.Е., Ющенко Е.Л. Алгебра, языки, программирование. – К.: Наук. думка, 1974. – 328 с.
6. Глушков В.М., Бондарчук В.Г., Гринченко Т.А. и др. АНАЛИТИК-74, 79, 89, 93, 2000 // Кибернетика и системный анализ. – 1995. – № 5. – С. 127–157.
7. *Цейтлин Г.Е.* Введение в алгоритмику. – Киев: Сфера. – 1998. – 310 с.
8. *Грис Д.* Наука программирования. – М.: Мир, 1984.
9. *Кнут Д.Е.* Искусство программирования: в 4-х т. – 1968–2005.
10. *Турский В.* Методология программирования. – М.: Мир, 1980.

11. *Лавров С.С.* Синтез программ // Кибернетика. – 1982. – № 2. – С. 23–32.
12. *Кахро М.И., Тыугу Э.Х.* Инструментальная система программирования на ЕС ЭВМ (ПРИЗ). – М.: Финансы и статистика, 1981.
13. *Тыугу Э.Х.* Концептуальное программирование. – М.: Наука, 1984.
14. *Biorner D., Jones C.B.* The Vienna Development Methods (VDM): The Meta-Language // Computer Science. Lecture Notes. – V. 61. – Berlin–Heidelberg–New York: Springer-Verlag, 1978. – 215 p.
15. *Петренко А.К.* Венский метод разработки программ // Программирование. – 2001. – № 1. – С. 3–23.
16. The RAISE Language Group. The RAISE Spesification Language. BCS Partner Series. – Prentice Hall, 1982. – 397 p.
17. *Агафонов В.Н.* Спецификации программ: понятийные средства и их организация. – Новосибирск: Наука, 1987. – 240 с.
18. *Ершов А.П.* Научные основы доказательного программирования. – М.: Президиум АН СССР, 1986. – С. 10–19 (на звание академика).
19. Развитие парадигм программирования. Курс «Основы функционального программирования». – http://www.intuit.ru/department/pl/funcpl/153.html http://schum.kiev.ua/let/
20. *Редько В.Н.* Основания программологии // Кибернетика и системный анализ. – 2000. – № 1. – С. 3–27.
21. *Глушков В.М.* Теория автоматов и формальные преобразования микропрограмм // Кибернетика. – 1965. – № 5. – С. 1–10.
22. *Капитонова Ю.В., Летичевский А.А.* Методы и средства алгебраического программирования // Кибернетика. – 1993.– № 3. – С. 7–12.
23. *Летичевский А.А., Маринченко В.Г.* Объекты в системе алгебраического программирования // Кибернетика и системный анализ. – 1997. – № 2. – С. 160–180.
24. *Letichevsky A.A., Gilbert D.R.* A General Theory of Action Language // Кибенетика и системный анализ. – 1998. – № 1. – С. 16–36.
25. *Редько В.Н.* Композиционная структура программологии // Кибернетика и системный анализ. – 1998. – № 4. – С. 47–66.
26. *Никитченко Н.С.* Композиционно-номинативный подход к уточнению понятия программы // Проблемы программирования. – 1999. – № 1. – С. 16–31.
27. *Редько В.Н.* Экспликативное программирование: ретроспективы и перспективы // Проблемы программирования. – 1998. – № 2. – С. 22–41.
28. *Лаврищева Е.М., Грищенко В.Н.* Связь разноязыковых модулей в ОС ЕС. – М.: Финансы и статистика, 1982. – 136 с.
29. *Лаврищева Е.М., Грищенко В.Н.* Сборочное программирование. – Киев: Наукова думка, 1991. – 269 с.
30. *Лаврищева Е.М., Грищенко В.Н.* Сборочное программирование. Основы индустрии программных продуктов. – Киев: Наукова думка, 2009. – 371 с.
31. *Лаврищева Е.М.* Software Engineering компьютерных систем. Парадигмы, технологии, CASE-средства программирования. – Киев: Наукова думка, 2013. – 284 с.
32. Автоматное программирование. – http://ru.wikipedia.org/wik