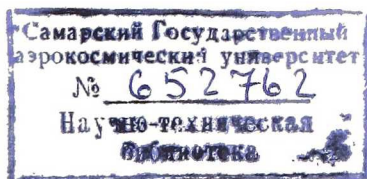


Министерство общего и профессионального образования
Российской Федерации
Самарский государственный аэрокосмический университет
имени академика С.П.Королева

А. Н. КОВАРЦЕВ

АВТОМАТИЗАЦИЯ РАЗРАБОТКИ И ТЕСТИРОВАНИЯ ПРОГРАММНЫХ СРЕДСТВ



УДК 681.3:519.6

ББК 32.97

С 40

Рецензент заслуженный работник высшей школы РФ, д-р техн. наук, проф. С.А. Прохоров

Коварцев А.Н.

Автоматизация разработки и тестирования программных средств

Самар. гос. аэрокосм. ун-т. Самара, 1999. 150 с.: ил.

ISBN 5-7883-0073-8

Рассматриваются проблемы автоматизации разработки и тестирования программных средств (ПС), используемых в САПР сложных технических систем на основе технологии графосимволического программирования (ГСП), обеспечивающей визуальный стиль программирования в процессе создания или модификации ПС. Технология ГСП позволяет исключить большинство видов дефектов и ошибок при создании или модификации ПС и их компонент и поддерживает такие этапы жизненного цикла ПС как проектирование, кодирование, отладка, документирование и сопровождение. Изложены методы автоматизации функционального тестирования вычислительных модулей, а также тестирования структур граф-программ, выполненных в рамках технологии ГСП. Значительное внимание уделено методам решения многокритериальной задачи принятия решений в условиях неопределенности исходной проектной информации. Предложен оригинальный метод построения Парето-оптимального множества оценок и критериев оценки риска невыполнения условий технического задания, используемый при выборе проектных параметров авиационных ГТД. Приводятся примеры использования ГСП для разработки ПС и выбору рациональных параметров ГТД на начальных этапах его проектирования.

Предназначена для специалистов, связанных с разработкой, эксплуатацией и сопровождением сложных программных средств.

ISBN 5-7883-0073-8

© А.Н. Коварцев, 1999

© Самарский государственный аэрокосмический университет, 1999

ОГЛАВЛЕНИЕ

ПРЕДИСЛОВИЕ.....	5
глава 1. Теоретические основы технологии графосимволического программирования (ГСП).....	7
1.1. Концептуальная модель ГСП.....	7
1.2 Базовые модули и типы данных	12
1.2.1. Базовые модули. Базисы	12
1.2.2. Типы базовых модулей	15
1.2.3. Типы данных	16
1.3. Объекты технологии ГСП	18
1.3.1. Акторы	19
1.3.2. Предикаты	21
1.3.3. Агрегаты	22
1.3.4. Объекты типа “in line”	23
1.4. Принципы организации управления вычислительным процессом в ГСП	24
1.5. Модель межмодульного информационного интерфейса технологии ГСП	29
1.5.1. Информационный интерфейс базовых модулей	32
1.5.2 Информационный интерфейс агрегатов	32
1.5.3. Информационный интерфейс EXE-модулей	34
глава 2. Конструирование объектов технологии ГСП	36
2.1. Конструирование объектов инкапсуляцией агрегатов ГСП	38
2.2. Использование рекурсии для конструирования объектов ГСП ...	41
2.3. Классификация данных объектов ГСП	51
2.3.1. Проблема классификации данных агрегатов	51
2.3.2. Декомпозиция агрегатов. Алгебра трехзначной логики выделения классификационных признаков	53
2.3.3. Построение множества вычислительных маршрутов	56
2.3.4. Сжатие числа операций процедуры классификации данных	60
2.3.5. Алгоритм классификации данных на основе частичного перебора маршрутов ориентированного графа.....	62
2.3.6. Эффективность алгоритма АЧП	65
2.4. Структурная оптимизация агрегатов	68
глава 3. Автоматизация тестирования объектов технологии ГСП	73
3.1. Стохастическое тестирование базовых модулей	74
3.1.1 Метод независимых испытаний	76
3.1.2 Адаптивный алгоритм тестирования акторов	84
3.2. Тестирование агрегатов технологии ГСП.....	89

3.2.1. Проверка корректности организации ветвлений из вершин графа.	90
3.2.2. Тестирование структуры агрегата	95
3.3. Оценка надежности объектов в технологии ГСП	99
3.3.1. Оценка надежности акторов	100
3.3.2. Оценка надежности агрегатов	103
Глава 4. Использование технологии ГСП для построения программного обеспечения САПР	108
4.1. Примеры использования технологии ГСП для автоматизации начальных этапов проектирования авиационных двигателей	108
4.1.1. Многокритериальная задача оптимизации при выборе параметров сложных технических систем	109
4.1.2. Аппроксимация множества эффективных оценок	117
4.1.3. Неопределенность в задачах выбора параметров рабочего процесса авиационных ГТД	123
4.1.4. Использование технологии ГСП для автоматизации разработки ГП начальных этапов проектирования авиационных ГТД.....	131
4.2. Исследование погрешностей вычислительных алгоритмов	138
4.2.1. Исследование сходимости численных методов решения системы нелинейных уравнений	139
4.2.2. Оценка инструментальной погрешности ЭВМ	143
Список использованных источников	146

ПРЕДИСЛОВИЕ

Уровень развития современного общества таков, что практически не существует ни одной из его сфер деятельности, где бы не использовались компьютерные технологии. Последнее порождает большой социальный спрос на одну из ее основных компонент - *программное обеспечение*.

В то же время с ростом сложности современных программных продуктов все более актуальной становится *задача повышения производительности труда разработчиков* программного обеспечения. Именно в связи со сложностью программных продуктов на более приоритетное место постепенно перемещается *проблема обеспечения надежности* разрабатываемых ПП.

Опыт создания программных комплексов в интересах военно-промышленного комплекса, например, таких как САПР ГТД или ЛА, стимулировал исследования по поиску путей уменьшения сроков разработки или модификации ПО САПР и повышения ее надежности. Особенностью данной предметной области является то обстоятельство, что математическое и программное обеспечение в САПР остается без изменений очень непродолжительное время. В САПР цели и методы исследования как правило эволюционируют по мере познания объекта исследования. Возможность "быстрой" и "безболезненной" модификации ПО для САПР является принципиальным моментом, существенно сокращающим сроки разработки новых изделий и повышающих их качество.

Решение перечисленных выше проблем лежит в плоскости комплексной автоматизации полного жизненного цикла ПП, включая проектирование, разработку, тестирование, испытание и сопровождение программ подобно тому, как это принято в САПР сложных технических систем.

С момента осознания необходимости применения системного подхода к программированию и принципа структурного программирования, провозглашенного Дейкстрой, теория программирования "прошла" замечательный путь от создания методов и автоматизированных средств проектирования ПП, разработки языков четвертого поколения, автоматизированных средств документирования ПП до разработки интегрированных платформ программирования, включающих известные средства автоматизации разработки различных этапов жизненного цикла ПП.

Вероятно, настал момент переосмысления способа кодирования алгоритмов ПП с учетом "человеческого фактора". Последнее позволяет выделить проблему разработки визуальных форм представления алгоритмов программы, информационных межмодульных интерфейсов, результатов тестирования программы и других ее компонент, которые соответствовали бы образному процессу мышления человека.

Решение этой проблемы связано с необходимостью проведения ревизии традиционных способов представления кодов программы, декомпозиционного

расслоения таких компонент ПП как структурное описание, организация управления и обмен данными между модулями ПП, а также создания методологической основы для реализации “визуального” стиля программирования и тестирования ПП на различных этапах его жизненного цикла.

Развитие методов автоматизации программирования происходило по многим направлениям: это и автоматическое порождение кодов программ, и структурная оптимизация ПП, однако наиболее ощутимые результаты появлялись всякий раз при смене парадигмы программирования, т.е. при изменении концепции структуризации в программировании. Среди наиболее известных технологий программирования следует указать на такие как: модульное программирование, логическое программирование, объектно-ориентированное программирование, конфигурационное программирование. В настоящее время утверждается новый визуальный подход к решению проблемы автоматизации программирования, ориентированный на максимальное приближение технологии программирования к образному способу мышления человека.

Предлагаемая читателю книга описывает один из вариантов решения проблемы автоматизации разработки и тестирования программных средств (ПС) главным образом для тех предметных областей, где используются “численные” программные модули (АСНИ, САПР и т.д.).

Технология ГСП это технология проектирования и кодирования сложных программных продуктов, базирующаяся на графическом способе представления алгоритмов разрабатываемых программ, преследующая цель полной или частичной автоматизации процессов проектирования, кодирования и отладки программных продуктов. В ней для класса вычислительных программных продуктов частично или полностью решены задачи автоматизации различных этапов жизненного цикла ПП, таких как структурная оптимизация ПП, тестирование и проектирование ПП. В отличие от существующих технологий реализована более глубокая структурная декомпозиция типовых компонент программ.

* * *

Неоценимую помощь в работе над книгой оказали А.В. Баландин, О.П. Солдатова, К.А. Кудрин, В.С. Кузьмичев. Всем своим коллегам автор глубоко благодарен. Особую признательность хочется выразить С.А. Прохорову, который поддерживал автора на всех этапах работы над рукописью и сделал множество важных замечаний. Автор благодарен ректору СГАУ В.А. Сойфери за поддержку и доверие.

ТЕОРЕТИЧЕСКИЕ ОСНОВЫ ТЕХНОЛОГИИ ГРАФОСИМВОЛИЧЕСКОГО ПРОГРАММИРОВАНИЯ

Вопросы повышения эффективности и качества труда программиста в настоящее время приобретают чрезвычайную важность в связи с происходящими в мировом сообществе процессами массовой информатизации. Современные изящные программные средства достигают более высокой эффективности за счет автоматизированной поддержки разных этапов жизненного цикла разрабатываемого программного обеспечения (ПО).

Впечатляющие успехи, достигнутые в области совершенствования аппаратных средств, в частности технических средств отображения информации, отразились и на программных средствах, используемых при создании ПО. В настоящее время утвердился термин “визуального программирования”, под которым понимается широкое использование графического способа изображения информации. Внешний интерфейс современных инструментальных систем и языков программирования широко использует многооконную технологию отображения информации в стиле Windows, а в самих системах появились развитые инструментальные средства создания и редактирования всех типов ресурсов Windows. Последнее обстоятельство в значительной степени автоматизирует процесс разработки ПО с развитым интерфейсом взаимодействия с пользователем.

Однако, несмотря на развитые средства и формы визуального представления информации, стиль программирования в таких системах остается текстовым, если не считать так называемые “визарды”, предназначенные для генерации приложений.

Создание визуального (образного) стиля разработки ПО является основным мотивом разработки технологии графосимволического программирования. В главе вводятся основные определения, используемые в технологии графосимволического программирования, описываются принципы управления вычислительными процессами, специфицируются типы объектов ГСП и т.д. В целом описываемая в главе аксиоматика и составляет концепцию *технологии графосимволического программирования*.

1.1 Концептуальная модель ГСП

Технология графосимволического программирования в качестве средства автоматизации программирования исповедует два основополагающих принципа: *визуальную, графическую форму представления алгоритмов программ* и *других компонент их спецификации; принцип*

структурированного процедурного программирования. Перечисленные принципы, с одной стороны, это дань полувековому господству в среде программистов процедурной парадигмы программирования, с другой стороны, это связано с переходом к визуальной эпохе программирования.

В качестве методологической основы для представления алгоритмов в работе используется модель объекта с дискретными состояниями [9]. Основу такой модели составляет идея, что для любого объекта программирования тем или иным способом можно выделить конечное число состояний, в которых он может пребывать в каждый момент времени. Тогда развитие вычислительного процесса можно ассоциировать с переходами объекта из одного состояния в другое. В математике такая концепция в качестве способа абстрагирования плодотворно используется достаточно давно: марковские цепи, теория массового обслуживания, теория формальных грамматик и автоматов и т.д.

Для уточнения понятия *состояния*, используемого в работе, определимся с принятой в ГСП концепцией модели алгоритма. Можно выделить следующие три основных типа универсальных алгоритмических моделей:

Первый связывает понятие алгоритма с наиболее традиционными понятиями математики - вычислениями и числовыми функциями. Наиболее известная и изученная модель такого типа - *рекурсивные функции*.

Второй основан на представлении об алгоритме как о некотором детерминированном устройстве, способном выполнять в каждый отдельный момент лишь примитивные операции. Одним из многочисленных представителей этого типа является машина Тьюринга.

Третий алгоритмических моделей это преобразование слов в произвольных алфавитах, в которых элементарными операциями являются подстановки. Среди моделей этого типа наиболее известны канонические системы Поста, нормальные алгорифмы Маркова и т.д.

Для технологии графосимволического программирования наиболее подходит первый тип формализации понятия алгоритма, когда произвольная программа интерпретируется некоторой вычислимой функцией

$$A : in(D) \rightarrow out(D),$$

где $in(D)$ - множество входных данных программного модуля A , $out(D)$ - множество выходных (вычисляемых) данных программного модуля A .

Наиболее строго данная концепция исследована для вычислительной модели, названной *машиной Колмогорова* [54]. Там же можно найти едва ли не единственное формальное определение понятия *состояния* вычислительного процесса. По Колмогорову *состояния* - суть конструктивные объекты [40], под которыми в случае технологии графосимволического программирования можно понимать ансамбли конкретизаций структур данных (входных или вычисляемых), используемых в алгоритме. На каждом шаге итерации реализуется переработка текущего состояния структур данных D в

новое состояние D^* с помощью некоторой локальной функции $D^* = f_k(D)$. Процесс переработки $D^0 = D$ в $D^1 = f_{k_1}(D^0)$, D^1 в $D^2 = f_{k_2}(D^1)$ и т.д. продолжается до тех пор, пока не появится сигнал о получении решения.

Через S обозначим множество состояний $S = \{S_1, S_2, \dots, S_n\}$ некоторого объекта программирования O . Для O определим множество локальных вычислимых функций непосредственной переработки данных $\bar{S} = \{f_1, f_2, \dots, f_m\}$. Здесь под вычислимой функцией будем понимать функцию, для вычисления которой существует алгоритм. Каждому из состояний S_i поставим в соответствие локальную вычислимую функцию f_{k_i} . В общем случае $m \leq n$. На машинах Колмогорова алгоритм описывается с помощью оператора непосредственной переработки Ω_r [54]. Несколько упрощая ситуацию, устройство оператора Ω_r можно интерпретировать как композицию локальных вычислимых функций (у Колмогорова локальных операторов) $\Omega_r(D) = f_{i_1}(f_{i_2}(f(\dots), \dots), \dots, f_{i_n}(D) \dots)$. Такой способ абстрагирования удобен для теоретических исследований понятия алгоритма как самостоятельного понятия, но совершенно непригоден для программирования. Для композиции локальных функций удобно пользоваться метрической схемой - так называемым *графом состояний*.

Определим *граф состояний* G как ориентированный помеченный граф, вершины которого - суть состояния, а дугами отмечаются переходы системы из состояния в состояние. Каждая вершина графа помечается соответствующей локальной вычислимой функцией f_k . Одна из вершин графа, соответствующая начальному состоянию, объявляется начальной вершиной и, таким образом, граф оказывается инициальным. Дуги графа проще всего интерпретировать как *события*. С позиций данной работы *событие* - это изменение состояния объекта O , которое влияет на развитие вычислительного процесса.

На каждом конкретном шаге работы алгоритма в случае возникновения близости, когда из одной вершины исходят несколько дуг, соответствующее *бытие* определяет дальнейший ход развития вычислительного процесса алгоритма. Активизация того или иного события так или иначе зависит от состояния объекта, которое в свою очередь определяется достигнутой конкретизацией структур данных D объекта O .

Для реализации *событийного управления* на графе состояний G введем множество предикативных функций $P = \{P_1, P_2, \dots, P_l\}$. Под предикатом будем понимать логическую функцию $P_i(D)$, которая в зависимости от значений данных D принимает значение равное 0 или 1. Дугам графа G поставим в соответствие предикативные функции. Событие, реализующее переход $S_i \rightarrow S_j$ на графе состояний G , инициируется, если модель объекта O

на текущем шаге работы алгоритма находится в состоянии S_i и соответствующий предикат $P_{ij}(D)$ (помечающий данный переход) истинен.

В общем случае предложенная концепция (без принятия дополнительных соглашений) допускает одновременное наступление нескольких событий, в том случае когда несколько предикатов, помечающих дуги (исходящих из одной вершины), приняли значение истинности. Возникает вопрос: на какое из наступивших событий объект программирования должен отреагировать в первую очередь? Традиционное решение этой проблемы связано с использованием механизма приоритетов. В связи с чем все дуги, исходящие из одной вершины, помечаются различными натуральными числами, определяющие их приоритеты. Отметим, что принятое уточнение обусловлено ресурсными ограничениями, свойственными однопроцессорной ЭВМ.

Определим универсальную алгоритмическую модель технологии графосимволического программирования четверкой $\langle D, \mathfrak{F}, P, G \rangle$, где D - множество данных (ансамбль структур данных) некоторой предметной области (для конкретного объекта программирования O структуры его данных $D \in \mathcal{D}$); $\mathfrak{F}$ - множество вычислимых функций некоторой предметной области; P - множество предикатов, действующих над структурами данных предметной области D ; G - граф состояний объекта O .

Представленная алгоритмическая модель не претендует на право считаться формализацией понятия "алгоритм", хотя и является универсальной, поскольку допускает описание любых алгоритмов. Универсальность модели обосновывается универсальностью алгоритмических языков, использующих управляющие структуры типа "цепочка" и цикл "пока", а также сводимость предложенной алгоритмической модели к универсальной машине Тьюринга.

Таким образом, в технологии графосимволического программирования в качестве универсальной алгоритмической модели предлагается использовать абстрактную модель $\langle D, \mathfrak{F}, P, G \rangle$, основанную на *графе состояний*.

Граф в данном случае заменяет текстовую (вербальную) форму описания алгоритма программы, при этом:

- 1). Реализуется главная цель представление алгоритма в визуальной (графосимволической) форме.

- 2). Происходит декомпозиционное расслоение основных компонентов описания алгоритма программного продукта. Так структура алгоритма представляется графом G , элементы управления собраны во множество предикатов P и, как правило, значимы не только для объекта O , но и для всей предметной области. Спецификация структур данных, а также установки межмодульного информационного интерфейса по данным "пространственно" отделена от описания структуры алгоритма и элементов управления.

Предложенная алгоритмическая модель $\langle D, \mathcal{I}, P, G \rangle$, в конечном счете, описывает некоторую вычислимую функцию $f_G(D)$ и в этом смысле может служить “исходным материалом” для построения алгоритмических моделей других программ. Последнее означает, что технология ГСП допускает построение иерархических алгоритмических моделей. Уровень вложенности граф-моделей в ГСП не ограничен.

Структура алгоритмической модели $\langle D, \mathcal{I}, P, G \rangle$ во многом зависит от выбранного способа декомпозиционного расслоения объекта программирования на множество состояний S и множество событий, определяемых предикативными функциями P . В каждой конкретной предметной области эта задача решается индивидуально и, как правило, не вызывает затруднений.

Рассмотрим конкретный пример построения граф-модели.

При исследовании операций часто приходится сталкиваться с анализом работы своеобразных систем, называемых *системами массового обслуживания* (СМО). Примерами таких систем могут служить: телефонные станции, справочные бюро, фондовые биржи, системы передачи и распределения информации и т.д. Изучение таких систем в полном объеме возможно только с использованием цифровой имитации. В качестве примера рассмотрим двухканальную СМО с отказами.

Множество состояний в данном случае можно определить традиционно (как это принято в теории массового обслуживания) по числу занятых каналов. Состояния будут:

- S_{00} - все каналы свободны,
- S_{10} - занят первый канал, второй свободен,
- S_{01} - занят второй канал, первый свободен,
- S_{11} - заняты оба канала.

Можно выделить три события, переводящие систему из одного состояния в другое:

- E_0 - приход очередной заявки,
- E_1 - завершение обслуживания заявки на первом канале,
- E_2 - завершение обслуживания заявки на втором канале.

В зависимости от целей исследования каждому состоянию можно поставить в соответствие вычислимую функцию, содержанием которой может быть: подсчет заявок, получивших отказ, сбор статистических данных по временам обслуживания заявок в СМО. наконец, имитация выполнения некоторых специфических действий, связанных с функционированием того или иного канала. Не вдаваясь в детали, установим следующее соответствие между состояниями и функциями: $S_{00} - f_0$; $S_{10} - f_1$; $S_{01} - f_2$; $S_{11} - f_3$. Событиям $E_0 - E_2$ поставим в соответствие предикаты $P_0 - P_2$. Содержание предикатов зависит от степени детализации процессов, происходящих в СМО. В простейшем случае это может быть простое “бросание кости” в соответствие с интенсивностью

потока заявок λ и интенсивностями обслуживания μ_1 и μ_2 . В более сложных случаях, возможно моделирование интервалов времени между событиями, связанных с приходом заявок или их обслуживанием.

Тогда граф-модель двухканальной СМО можно представить так, как это показано на рис 1.1.

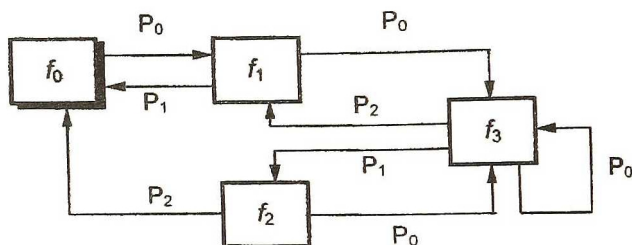


Рис. 1.1. Граф-модель двухканальной СМО

1.2 Базовые модули и типы данных

В качестве исходного строительного материала в технологии ГСП выступают две категории: базовые модули и типы данных. Базовые модули представляют собой перечень локальных вычислимых функций, на основе которых в конечном итоге порождаются все объекты технологии ГСП (акторы, агрегаты и предикаты). Типы данных описывают синтаксический и семантический аспекты строения данных, используемых в базовых функциях, а также и в объектах технологии ГСП.

Если вкладывать в понятие программирование традиционный смысл, то в ГСП оно начинается на этапе конструирования новых граф-программ из имеющегося набора вычислимых функций, точно так же, как программы, написанные на алгоритмических языках программирования, состояются из конечного набора операторов языка. Порождение первоначального множества вычислимых функций (базовых модулей) производится на любом из существующих языков программирования, например, языке C++.

1.2.1 Базовые модули. Базисы

Программирование представляет собой сложный интеллектуальный процесс, естественное развитие которого в 70-х годах привело к возникновению нового понятия методологии программирования. Научное направление, символизируемое Дейкстрой, Виртом, Хоаром, в рамках

структурного программирования развивает теорию программирования как методологию понимания сложных проблем [17]. Обилие "технических" трудностей в программировании возникает всякий раз при решении конкретных практических задач, которые по Б.Мейеру [43] связаны одновременно с ограничениями, внутренне присущими человеческому уму, и с общей проблемой передачи идей и разделения труда. Основной задачей методологии программирования является сведение при помощи систематической декомпозиции задач очень большой сложности к комбинации простых задач, чтобы при этом синтез решений было легко реализовать. Для того, чтобы построить программу, необходима стратегия декомпозиции. Однако остается уточнить, каким должен быть основной элемент в построении программы, т.е. единица декомпозиции.

Наиболее простой классический ответ заключается в том, чтобы рассматривать элемент программы, определенный своими входами и выходами, т.е. подпрограмму. Подобная концепция вводит понятие *модуля*.

В литературе до сих пор не сложилось окончательное определение понятия модуля. Однако отчетливо выделяются два подхода к интерпретации этого понятия.

Первое - определяет модуль как замкнутую программную единицу с набором иницилирующих входов и указателей выходов, которую можно вызывать из любого другого модуля программы и отдельно компилировать.

Второе представление связывает понятие модуля с данными, которыми он оперирует. Данный подход во многом обусловлен решением сложной проблемы распределения ответственности: какой программный модуль должен обеспечивать управление набором данных, используемых многими программными единицами? Эта проблема приводит к стратегии декомпозиции, основанной на понятии *модуль данных* в большей степени, чем на понятии *программный модуль* [59]. С этих позиций модуль есть *структура данных*, доступная извне при помощи некоторого набора программ (методов), и только при их помощи.

В этом случае модуль превращается в активный элемент (объект), обладающий всем множеством операций (методов) своего класса, а составление программы происходит в терминах взаимодействующих объектов, где активные элементы именно объекты, а не использующие их процессы. Таким образом, речь идет о необходимости считать программу не совокупностью процессов, обрабатывающих данные, а совокупностью активных "машин", взаимодействующих между собой [20]. На практике это направление получило распространение в так называемых объектно-ориентированных языках программирования.

Таким образом, перечисленные выше подходы к определению понятия *модуля* порождают две стратегии декомпозиции задачи построения программы и в конечном итоге альтернативные парадигмы программирования. Интересно, что в объектно-ориентированном программировании методология

программирования поставлена во главу угла, поскольку успех программирования в этом стиле во многом зависит от корректности выполнения операции “расслоения свойств” предметной области, для которой разрабатывают программы [33]. В объектно-ориентированном подходе программная таксономия проводится всегда либо осознанно, либо стихийно. Методы структурного программирования настоятельно рекомендуют проводить предварительный модульный анализ предметной области, однако на практике благие намерения редко реализуются.

В ГСП отмечают два аспекта, объединяющие разные трактовки модуля: выделение в обособленную самостоятельную единицу каких-то действий, функций и данных; выделение в самом модуле его внешности (интерфейса) - той части, посредством которой модуль связан с внешним миром (другими модулями, операционной средой и т.п.).

Будем понимать под модулем *независимую программную единицу, реализующую определенную функцию в процессе преобразования некоторого агрегата данных.*

Предложенное определение модуля достаточно формально и не раскрывает содержательной сущности модуля. Под это определение попадают практически любые программы, оформленные в виде процедуры. Одна из составляющих компонент методологии программирования - модульный анализ пытается более четко очертить сущностные аспекты понятия модуля путем перечисления его основополагающих свойств.

Центральная концепция модульности относительная независимость [30], характеризуемая двумя критериями: прочностью (связностью) и сцеплением, т.е. модуль должен реализовывать по возможности единственную функцию (высокая прочность) и иметь как можно меньше связей с другими модулями (низкое сцепление). Кроме того, модуль должен быть “актуальным”, т.е. использоваться по возможности в большем числе задач и “содержательным” (функциональная самостоятельность), решающим нетривиальную подзадачу, с тем чтобы решение задач можно было представить ограниченным набором модулей.

Довольно часто по разным причинам ограничивают размеры модуля, например, по соображениям удобства редактирования и наглядности или из соображений сокращения времени трансляции и т.д. В технологии ГСП размеры базовых модулей ограничены соображениями возможности проведения полного цикла тестовых испытаний.

Однако для ГСП наиболее существенным является свойство независимости (ортогональности) базовых моделей [15]. Ортогональность подразумевает однозначное, исключаящее дублирование, распределение функций между модулем и остальной программой. Для строгого определения ортогональности двух модулей B_i и B_j требуется ввести некоторую оценочную функцию $\varphi(B_i, B_j)$, являющуюся мерой независимости этих модулей и

принимая значения на $[0,1]$ (0 - в случае полной независимости модулей и 1 при их совпадении). Оценочная функция $\varphi(B_i, B_j)$ играет роль скалярного произведения.

Будем говорить, что система базовых модулей $B_1, B_2, \dots, B_n$ *независима*, если никакой из модулей этого множества не может быть представлен как суперпозиция оставшихся модулей.

Система модулей $B_1, B_2, \dots, B_n$ обладает свойством *полноты* в некоторой предметной области, если любой алгоритм предметной области может быть представлен как суперпозиция базовых модулей.

Полная система независимых модулей $B_1, B_2, \dots, B_n$ некоторой предметной области образует базис.

Ортогональность базовых модулей в технологии ГСП крайне желательна, поскольку, с одной стороны, исключает дублированное распределение функций между модулями, с другой стороны, позволяет перевести программирование из плоскости традиционного языкового программирования в плоскость образного конструирования граф-программ. Обеспечение "повальной" базисности базовых модулей обычно оказывается недостижимым идеалом. Однако практически любые шаги, предпринимаемые разработчиком в этом направлении, не пропадают даром, так как не только продвигают технику программирования в сторону более технологичного графосимволического способа, но и повышают такое полезное качество как надежность разрабатываемого программного обеспечения.

Принцип ортогонализации, в известной степени, используется при построении библиотек прикладных программ в современных языках программирования. Так, базис содержится в множестве функций обработки символьных строк языка C++, в библиотеке графических программ, в библиотеке математических функций и т.д.

1.2.2. Типы базовых модулей

В [20] А.В. Замулиным вводится понятие *типа функции* как обобщение понятия типа операции и определяется как множество отображений из области определения функции в область результата. Причем областью определения функции считается декартово произведение множеств значений нескольких типов данных (типов формальных параметров), областью результатов множество значений некоторого одного типа данных. При этом тип функции изображается $T_1, T_2, \dots, T_n \rightarrow T_p$, где $T_1, T_2, \dots, T_p$ типы формальных параметров и результата вычислений рассматриваемой функции.

С каждым типом функции (подпрограммы) связывают две операции: создания функции и аппликации функций к своим аргументам. Первая операция в языках программирования связана с изображением функции в виде: заголовок функции плюс тело функции. Вторая операция - в виде обращения к функции.

В теории программирования подпрограммы - одно из фундаментальных средств абстрагирования и искусственного расширения возможностей языка, однако правила реализации первой из перечисленных выше операций - составления тела подпрограммы, обычно синтаксически неотличимы от правил кодирования основного текста программы. Последнее достигается за счет введения понятия формальных параметров. В то время как между основной программой и подпрограммами имеет место существенная разница. При разработке основной программы преследуется конкретная практическая цель, например, произвести расчеты некоторого технического устройства, физического явления, математических формул и т.д. При этом данным, используемым в программе "придается" вполне конкретный смысл (давление, скорость, пропускная способность и т.п.). В подпрограммах формальные параметры лишены конкретной смысловой нагрузки.

Действительно, в подпрограммах важен лишь тип параметра и порядок его использования, а "осмысление" назначения параметров возникает только после их аппликации к фактическим параметрам. Например, процедура, реализующая формулу $A = B \cdot C$, в одной интерпретации типов данных вычисляет силу F по заданным ускорению a и массе m материальной точки ($F = a \cdot m$), в другой - путь S , по заданной скорости V и времени t ($S = V \cdot t$). Таким образом, подпрограмму можно рассматривать как описание абстрактного типа вида $B: T_1, T_2, \dots, T_n \rightarrow T_p$, в котором в теле подпрограммы реализуются операции над типами данных. В современных языках программирования такая интерпретация понятия подпрограммы присутствует незримо, например в языке C++ в заголовке подпрограммы необходимо указывать кроме имен формальных параметров их типы, а в предописании подпрограмм перечисляются только типы параметров.

В технологии ГСП исходные, модифицируемые данные и результаты вычислений базовых модулей размещаются в списке типов данных реализуемой функции, поэтому тип базового модуля определим как отображение типов данных из области определения на область их значений $B: T_{i_1}, T_{i_2}, \dots, T_{i_n} \rightarrow T_{j_1}, T_{j_2}, \dots, T_{j_m}$, которые реализуются в соответствии с принятыми в B операциями над типами данных.

1.2.3. Типы данных

Тип данных в технологии ГСП подразумевает традиционную трактовку типа данных T сигнатуры Σ , как пары: спецификации типа данных сигнатуры Σ и соответствующей ей реализации типа данных [20]. Здесь под сигнатурой понимается пара $\Sigma = \langle S, \Omega \rangle$, где S - множество имен-основ (имена базовых типов, базового языка программирования или производные от этих типов типы данных), а Ω есть $(S^* \times S)$ индексированное семейство множеств имен операций, S^* - множество всех цепочек элементов множества S . В таком

определении типа данных отражаются два аспекта: пользовательский, когда программист, составляя свою программу, видит тип как спецификацию; и машинный, связанный со способом реализации в ЭВМ обозначенного типа данных. Чтобы создать новый тип данного, необходимо построить спецификацию и сделать в ней необходимую реализацию. Для каждого типа данных имеется свой набор операций, гарантирующий “невыпадение” значений данных описанного типа из представленной спецификации.

Типизация данных полезна не только в гносеологическом смысле, как инструмент изучения теории программирования, но и в чисто прикладном аспекте, позволяя избежать большого количества ошибок, поскольку заставляет программиста точно определять все используемые им объекты и лучше контролировать свою программу.

В этом смысле чисто синтаксическую спецификацию типа данных (семантический аспект типа сводится к проверке принадлежности значений данных своему типу) можно считать недостаточной.

В технологии ГСП понятие типа данных расширено понятием интерпретации данного. Действительно во многих предметных областях чисто “языковое” представление типа данного является неполным. Например, в физике в формуле $F = at$ все три компонента формулы F , a , t имеют естественный языковый тип `double` (вещественные числа) с определенными на этом типе операциями $(+, -, *, /)$, однако каждое из перечисленных данных имеет самостоятельную (смысловую) интерпретацию: F сила [н], a ускорение [м/сек²], t - масса [кг]. Очевидно, что передача в подпрограмму вместо ускорения a (тип `double`) скорости v (тип `double`), не вызовет синтаксической ошибки, но приведет к неправильному исполнению программы. Следует отметить, что такого рода ошибки обычно очень сложно распознаются.

В качестве концептуальной основы введения типа интерпретации данного в ГСП предлагается использовать теорию размерности.

Определим множество основ типа интерпретации данных S_{int} как множество образующих типов интерпретации и их производных. Например, для физических задач это множество имеет вид:

$$S_{\text{int}} = \{[м], [кг], [сек], [м/сек], \dots\}.$$

Множество операций в простейшем случае может быть представлено естественными операциями над типами интерпретации $\Omega_m = \{+, -, *, /\}$. При описании типов интерпретации данных базовых модулей целесообразно ввести понятие множества аксиом A_{int} , действующих над типами интерпретаций. Множество A_{int} состоит из замкнутых σ -формул. Например, для формулы $F = at$ аксиомой типобразования служит формула: $T_1 = T_2 * T_3$.

Тип интерпретации данного определим как $T_{int} = \langle \Sigma_{int}, A_{int} \rangle$, где $\Sigma_{int} = \langle S_{int}, \Omega_{int} \rangle$. Таким образом, под типом данного в технологии ГСП понимается пара $T = \langle \Sigma, \Sigma_{int} \rangle$.

Если определение некоторого типа данного не расширено семантическим аспектом - описанием типа интерпретации, то считается, что данный тип имеет "пустую" размерность ($[]$). При верификации используемых типов данных в ГСП первоначально проверяется синтаксическая составляющая описания типа, а затем сравниваются типы интерпретации. Спецификация данных в технологии ГСП реализована в виде совокупности таблиц информационного фонда системы. В отдельную таблицу, называемую архивом типов данных, сведены описания всех необходимых атрибутов типов данных, начиная с имени типа, описания родового языкового типа или его редукции, описания типа интерпретации и заканчивая описанием области возможных значений (домена). Базовые модули рассредоточены по множеству библиотек специализированных программ (базисам технологии ГСП).

1.3 Объекты технологии ГСП

В технологии ГСП в качестве программных единиц рассматриваются *объекты*. По способу порождения и функциональному назначению различают три типа объектов: *акторы, агрегаты и предикаты*. Все они имеют конкретный содержательный смысл и действуют в рамках предметной области программирования (ПОП).

Под предметной областью программирования будем понимать некоторую среду программирования, имеющую *общую цель* - разработку программного обеспечения автоматизации расчетов в некоторой области практических интересов (авиационные двигатели, бизнес, медицинские приборы и т.д.), *общую область данных* и *общую область знаний*. Понятие предметной области является понятием сферы деятельности программиста и одновременно его средой программирования.

В предметной области, как правило, заранее уже определен терминологический словарь данных (параметров, переменных или констант). Так например, в теории газотурбинных двигателей перечень параметров их обозначение и содержание регламентировано государственными стандартами, которые во многом унифицированы с международными стандартами. Аналогичное положение дел имеет место в самолетостроении, ракетостроении, радиоэлектронике и т.д.

Под предметной областью программирования в дальнейшем понимается *среда программирования, состоящая из общего набора данных (словарь данных) и набора программных модулей (словарь и библиотека программных модулей)*.

Поэтому программирование в рамках технологии ГСП начинается с утверждения и формирования, так называемого, *словаря данных* ПОП, который служит целям каталогизации данных ПОП, спецификации их семантики и областей значений.

Словарь данных представляет собой таблицу, в которой каждому данному присвоено уникальное имя, задан тип, начальное значение данного и краткий комментарий его назначения в ПОП.

Технология ГСП поддерживает жесткие стандарты на описание и документирование программных модулей, представление и поддержку информационного обеспечения программных модулей предметной области. Таким образом, для каждой предметной области строится единая информационная среда, позволяющая унифицировать проектирование написание программных модулей разными разработчиками.

Кроме словаря данных и каталога типов данных информационную среду определяют объекты ГСП. Под объектом понимается специальным образом построенный в рамках технологии ГСП программный модуль, выполняющий определенные действия над данными ПОП.

1.3.1. Акторы

Одним из объектов технологии ГСП является *актор*. Актор формируется из *базового модуля* путем привязки абстрактных типов данных базового модуля к данным предметной области. По сути дела *актор* порождается в результате аппликации базового модуля к своим аргументам (операция конкретизации над типом базового модуля, см. п. 1.2).

Актор производит те же действия, что и породивший его базовый модуль, но над конкретными данными ПОП. В отличие от базовых модулей, каждый актер является содержательным программным модулем, который выполняет понятные функции в рамках заданной предметной области. Актеры в технологии ГСП реализуют отображение над множеством данных предметной области: $A_k : D_k^{in} \rightarrow D_k^{out}$ где $D_k^{in} = \{d_1^{in}, d_2^{in}, \dots, d_n^{in}\}$ - множество входных данных актора A_k , $D_k^{out} = \{d_1^{out}, d_2^{out}, \dots, d_n^{out}\}$ - множество выходных данных актора A_k . Множества D_k^{in} и D_k^{out} образуют в совокупности полное множество данных некоторой предметной области (словарь данных): $D = D^{in} \cup D^{out}$

Один базовый модуль может породить множество акторов. В данном случае проявляется свойство параметрического полиморфизма базовых модулей технологии ГСП [55].

Между базовым модулем и актором осуществляется односторонняя связь типа "один ко многим". Каждый актер имеет свой прототип в виде базового модуля, а на основании каждого базового модуля можно построить один или несколько акторов. Это свойство полиморфизма объектов позволяет избежать

избыточности при порождении новых акторов, которые различаются между собой только привязкой по данным. Другими словами, на основе одного отлаженного и оттестированного базового модуля за счет механизма автоматизированной привязки по данным можно построить несколько корректных акторов, что позволяет значительно повысить степень надежности программирования в ГСП.

Порождение актора производится путем формирования, так называемого, *паспорта* объекта. Процедура паспортизации базового модуля заключается в установке соответствия между списком типов данных базового модуля и данными предметной области, таким образом, что каждому формальному параметру (типу данных) ставится в соответствие конкретное данное ПОП.

Соответствие между базовым модулем B_i и актором A_j порождает соответствие между подмножеством типов T_i данных и подмножеством самих данных D_j предметной области :

$$\begin{cases} B_i(T_i^{in}, T_i^{out}) \rightarrow A_j(D_j^{in}, D_j^{out}) \\ T_i = (T_i^{in}, T_i^{out}) \rightarrow D_j = (D_j^{in}, D_j^{out}) \end{cases}$$

В этом случае абстрактные операции над типами данных базового модуля превращаются в конкретные функциональные преобразования данных ПОП, т.е. формируется локальная вычислимая функция предметной области, например, термогазодинамический расчет компрессора ГТД.

Сформированное отношение (*паспорт* актора) оформляется как таблица БД (обозначим его $P(t,d)$) информационного фонда ГСП, содержащая перечень имен формальных параметров и соответствующих им имен данных ПО с указанием способа получения ими своих значений. По способу получения своих значений данные в паспорте делятся на три группы:

- 1) иницируемые (импортируемые) данные, которые должны принять значения до их использования объектом;
- 2) вычисляемые (экспортируемые) данные, которые впервые получают свои значения в процессе выполнения объекта;
- 3) модифицируемые (изменяемые) данные, которые образуются путем пересечения множеств иницируемых и вычисляемых данных.

При регистрации базового модуля в ПОП автоматически порождается дубликат базового модуля, имеющего стандартную форму информационного интерфейса:

<имя объекта>(<указатель на массив адресов данных ПОП>);

Тогда любой базовый модуль, разработанный на языке C++:

$B(t_1, t_2, \dots, t_n)$
 { тело базового модуля; }

в процессе регистрации превращается в модуль стандартного вида:

$B^*(void *P[])$
 { return($B(<тип\ t_1>P[0], <тип\ t_2>P[1], \dots, <тип\ t_n>P[n-1])$); }

Для стандартизированной формы базового модуля операция конкретизации типов данных сводится к процедуре формирования массива адресов данных $P[]$, в котором номер элемента массива соответствует порядковому номеру типа данных в базовом модуле, а значение элемента указывает на данное предметной области. Массив $P[]$ легко заполнить по паспортным данным актора, следовательно базовый модуль в стандартной форме + наполненный массив $P[]$ фактически определяют актор ПОП. На рис 1.2 показана схема порождения акторов расчета таких узлов авиационного двигателя, как вентилятор и компрессоры высокого и низкого давления путем привязки типов данных абстрактной вычислительной схемы, “заложенной” в соответствующем базовом модуле.

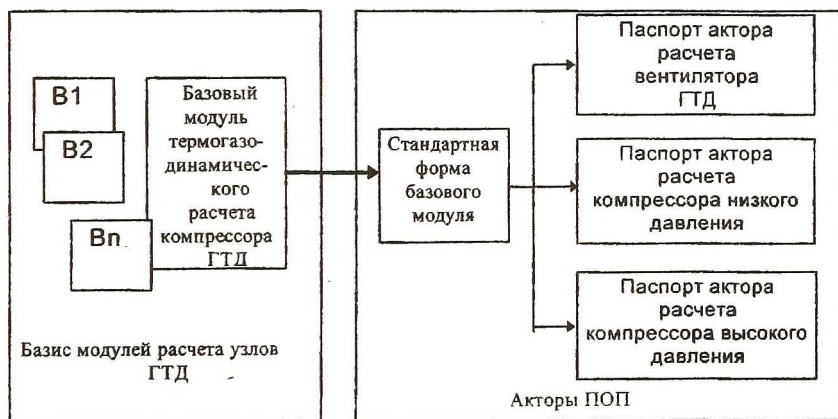


Рис 1.2 Порождение акторов в предметной области

1.3.2. Предикаты

В процессе реализации алгоритма в рамках технологии ГСП передача управления между объектами осуществляется с помощью управляющих объектов-предикатов. Формально предикат представляет собой отображение из множества данных предметной области на множество логических значений “истина” и “ложь”: $\Psi_k : (d_1, d_2, \dots, d_m) \rightarrow \{0, 1\}$. Отличие предиката от актора заключается в том, что предикат не может производить преобразование над данными, то есть все его данные являются входными: $(d_1, d_2, \dots, d_m) \in D^m$.

Технологически порождение предикатов ничем не отличается от процедуры порождения акторов. Первоначально строится базис абстрактных логических функций, действующих над типами данных. Множество предикатов ПОП строится из абстрактных логических функций в результате аппликации их типов данных к данным ПОП, т.е. в результате паспортизации типов данных логических функций. Рассмотрение *акторов и предикатов*, как различных категорий *объектов* ГСП, обусловлено не только особенностями реализуемых ими типов отображений, но и, что особенно важно, их ролевыми назначениями. Акторы, являясь локальными вычислимыми функциями, реализуют преобразование данных ПОП, в то время как предикаты, являясь функциями управления вычислениями, образуют базу знаний для всей предметной области, т.е. они общезначимы для любых программ, разрабатываемых в рамках ПОП.

1.3.3. Агрегаты

Объекты (акторы или предикаты) являются исходным материалом для графического программирования. Результатом графического программирования являются агрегаты. Агрегат создается в форме графа, в котором объекты ПОП играют роль вершин и дуг. Дуги - предикаты, а вершины - акторы или агрегаты. Дуги графа определяют передачу управления от одной вершины к другой.

Формально агрегат представляет собой помеченный ориентированный граф с входной (корневой) и несколькими выходными (концевыми) вершинами:

$$G = \{ F, \Psi, R \},$$

где $F = \{ A_1, A_2, \dots, A_n \}$ - множество акторов, которые являются вершинами графа, $\Psi = \{ \Psi_1, \Psi_2, \dots, \Psi_m \}$ - множество предикатов, которые представляют дуги графа, R - отношение над множествами вершин и дуг графа, определяющее способ их связи.

Корневой вершиной графа A_0 является такая вершина, из которой есть маршрут по графу в любую другую вершину, и которая помечена как корневая. Из этой вершины начинается выполнение алгоритма, реализованного агрегатом. Аналогично определяется конечная вершина A_n - это вершина, в которую есть маршрут из любой другой вершины, и которая не имеет исходящих дуг-предикатов. Концевых вершин на графе может быть несколько, если все они удовлетворяют поставленным условиям.

Развитие вычислительного процесса в агрегате происходит путем передачи управления из одной вершины в другую, начиная с корневой. Этот процесс может быть завершен по двум причинам: либо достигнута конечная вершина графа, из которой нет исходящих дуг, либо из текущей вершины отсутствуют разрешенные другими предикатами переходы в другие вершины. Если в процессе передачи управления сложилась ситуация, когда истинными

одновременно являются несколько предикатов, то управление будет передано по предикату, имеющему наибольший приоритет.

При таком подходе, между агрегатом в технологии ГСП и блок-схемой алгоритма существуют аналогии. Отличие заключается в том, что агрегат не имеет специальных управляющих блоков (условие и выбор), и передача управления всегда осуществляется посредством проверки предиката, который в частном случае может быть тождественно истинным. Это упрощает визуальный анализ алгоритма, за счет чего можно сократить число структурных ошибок. Например, ошибок, связанных с переусложненной структурой (неправильно вложенные циклы, неверная передача управления и т.п.), либо ошибок вызванных противоречиями в самом графе (непредусмотренные циклы).

В отличие от акторов и предикатов, которые полностью определяются своими паспортами, при порождении агрегата с помощью специального компилятора формируется текст нового объекта ПОП, который после трансляции заносится в библиотеку объектных модулей ПОП. Общие принципы структурной организации агрегата описаны в разделе 1.4.

1.3.4. Объекты типа "in line"

Опыт эксплуатации первой версии системы GRAPH показал, что достаточно часто возникает необходимость порождения небольших объектов, состоящих из одного - двух операторов базового языка программирования. Это всевозможные счетчики типа $I = I + 1$, отладочные печати, простые логические условия типа $A < B$, $A = 10$ и т.д. Разработка соответствующих базовых модулей с последующей паспортизацией до уровня объекта - дело неблагодарное. В связи с этим, в технологию ГСП была введена возможность порождения актора или предиката, минуя стадию разработки базовых модулей, которые получили название объектов типа "in line". Все модули "in line" жестко привязаны к данным ПОП, а стандартизированная форма программного модуля автоматически компилируется в процессе его порождения.

Итак, в наиболее общем виде, программирование в технологии ГСП состоит из следующих этапов:

1. Создание словаря данных ПО. На данном этапе создаются новые типы и структуры данных, а также происходит накопление словаря данных, где хранится информация обо всех переменных программы. Этот этап целесообразно реализовать с помощью удобного, гибкого многооконного интерфейса пользователя с информационным фондом системы.

2. Разработка базовых модулей. Это этап традиционного текстового программирования, на котором программист работает с исходными текстами программ, с использованием требований стандарта ГСП к оформлению этих текстов.

3.Создание объектов ПО. Этот этап производится автоматически после привязки формальных параметров базовых модулей к фактическим данным предметной области.

4.Конструирование агрегатов. На этапе графического программирования пользователь может создать графовый образ новой программы.

Разработанный и отлаженный агрегат, в свою очередь, может быть использован в качестве исходного материала при конструировании следующих агрегатов. Следовательно, в общем случае агрегат имеет иерархическую структуру.

Эффективность программирования в технологии ГСП возрастает по мере развития пользователем своей среды программирования. Доля текстового программирования с традиционной трудоемкой отладкой постепенно снижается и программирование перерастает в конструирование агрегатов из надежных программных модулей. Отладка при этом заключается только в корректировке структуры графа.

1.4. Принципы организации управления вычислительным процессом в ГСП

В технологии ГСП в зависимости от стиля программирования (последовательного программирования, последовательно-параллельного программирования, асинхронного программирования) используются различные схемы управления вычислительным процессом. Каждому стилю программирования соответствует собственная схема модели управления, при незначительных изменениях в синтаксисе представления граф-программ.

С помощью нестандартных схем управляются лишь части объектов, построенных средствами системы GRAPH и получивших название граф-объектов (агрегатов). Объекты, порожденные из базовых модулей (акторы), управляются стандартными методами, принятыми в базовом языке программирования. Синтез нового объекта (агрегата) происходит на основе его графического описания, представленного соответствующими структурами данных (хранящихся в информационном фонде системы GRAPH). Структура порождаемого объекта (см. рис. 1.3) состоит из автоматически компилируемой процедурной части агрегата и управляющего модуля GM. Процедурная часть агрегата содержит декларирующую часть агрегата, в которой описываются используемые в нем объекты, предикаты и данные. На основании описания граф-объекта заводятся соответствующие структуры данных, управляющие ходом вычислительного процесса.

Алгоритм управления вычислительным процессом реализован в отдельном модуле (GM), получившем название граф-машины. Процедурная часть агрегата индивидуальна для каждого объекта, в то время как граф-машина является универсальной программой, ответственной за ход выполнения вычислений для всех агрегатов. Вычислительный процесс развивается под управлением GM в соответствии с информацией, “заложенной” в управляющих структурах данных агрегата. В этом смысле

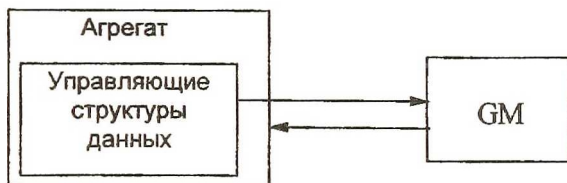


Рис. 1.3 Функциональная схема агрегата.

работа граф-машины напоминает функционирование машины Тьюринга, в которой лента заменена управляющими структурами данных.

Последовательное (процедурное) программирование в технологии ГСП реализуется механизмом, который в [16] определяется как механизм интеррогативного управления. В основе такого управления лежит представление о том, что граф-машина GM способна обеспечить постоянную проверку условий перехода граф-программы из одного состояния в другое. Логика развития вычислительного процесса в этом случае определяется логикой изменения значений системы предикативных функций, разрешающих или запрещающих переходы программы из одного состояния в другое.

В каждый конкретный момент времени граф-машина GM находится в одной из вершин графа (в состоянии выполнения соответствующего объекта граф-программы), которую будем называть текущей вершиной графа. Управляющая программа GM на каждом шаге её работы обеспечивает переход из текущей вершины A_k в вершину инцидентную ей, например, как это показано на рис.1.4 - в одну из вершин списка $A_{i_0}, A_{i_1} \dots A_{i_L}$. Переход возможен, если

соответствующий предикат $r_{i_k}^k$ принял значение “истина” (“1”). Во избежание неоднозначности при реализации перехода, в случае, когда одновременно несколько предикатов приняли значение “1”, дуги помечаются приоритетными индексами, причем чем меньше индекс, тем выше приоритет. Поэтому переход на графе будет реализован по дуге, для которой предикат принял значение “истина” и которая имеет наивысший приоритет. Для простоты положим, что верхний индекс в обозначении предиката устанавливает приоритет дуги.

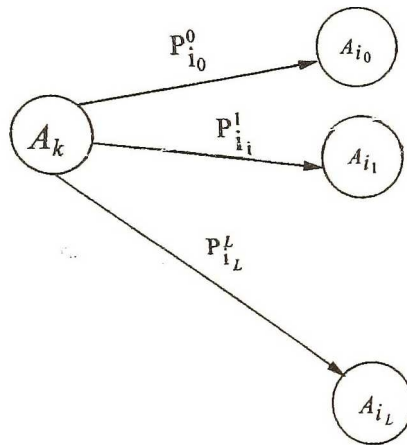


Рис. 1.4 Фрагмент граф-программы

Работу граф-машины ГМ можно интерпретировать с “блужданием” по описанию граф-программы из одной вершины в другую, которое может быть завершено по двум причинам:

1) достигнута конечная вершина графа, из которой нет исходящих дуг (нормальное завершение граф-программы);

2) из текущей вершины отсутствуют “разрешенные” предикатами переходы в другие вершины графа (ненормальное (аварийное) завершение программы).

Введем следующую систему обозначений:

Пусть A_{i_k} ($k=0,1,\dots,L$) объекты, в которые граф-машина способна перейти на очередном шаге ее работы из текущей вершины A_k по дуге, помеченной предикатом $P_{i_k}^k$.

Объекты в ГСП формально определяются как отображение

$A_k: in(D_k) \rightarrow out(D_k)$, где $in(D_k)$ - множество входных данных программного модуля A_k , $out(D_k)$ - множество выходных (вычисляемых) данных программного модуля A_k .

Предикаты $P_{i_k}^k: in(D_k) \rightarrow \{0,1\}$ - логические функции, действующие над конкретизацией данных ПО ($in(D_k)$) полученных к моменту их “проверки”

Обозначим $L_{Arc} = [P_{i_0}, P_{i_1}, \dots, P_{i_L}]$ список предикатов, помечающих дуги, исходящие из текущей вершины.

Введем две операции для построения списков вершин и списка дуг, инцидентных вершине A_k : $IncA(A_k)$ и $IncP(A_k)$.

Пустой список условимся обозначать $[]$.

Указатель на текущую вершину графа обозначим A_T .

Пусть Еггор - функциональный модуль граф-машины GM, выполняющий некоторый стандартный набор действий в случае возникновения ситуации ненормального завершения программы. При этом признак Er принимает значение 1.

Для описания работы алгоритма граф-машины GM воспользуемся выразительными возможностями описания программ в терминах Р-схем [6]. Алгоритм работы граф-машины GM в случае интеррогативного управления представлен 2-мя графами: move (A_T) и GM (см. рис. 1.5-1.6).

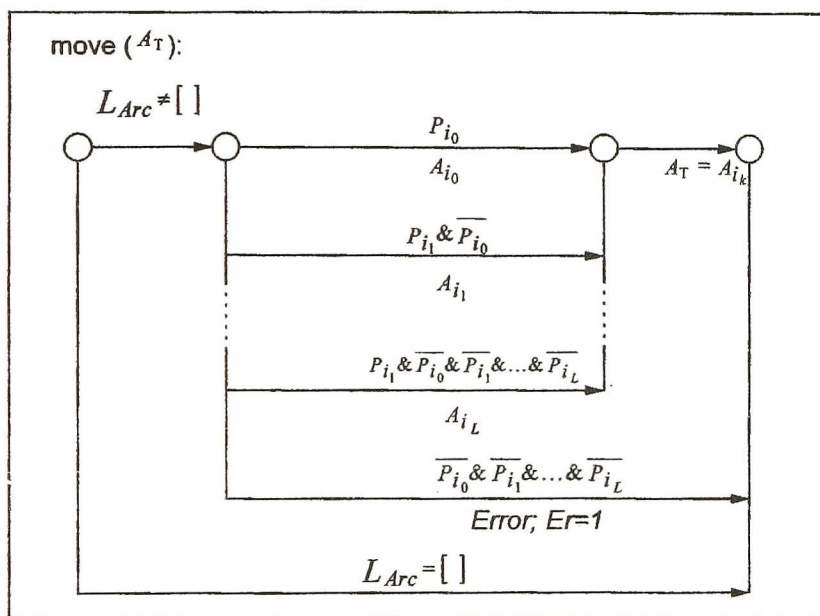


Рис. 1.5

В технологии Р-схем над дугами графовой схемы “надписываются” условия реализации (предикаты) одного или нескольких действий (программных модулей), которые “подписываются” под соответствующими дугами граф-схемы. Элементом  помечаются циклические блоки программы.

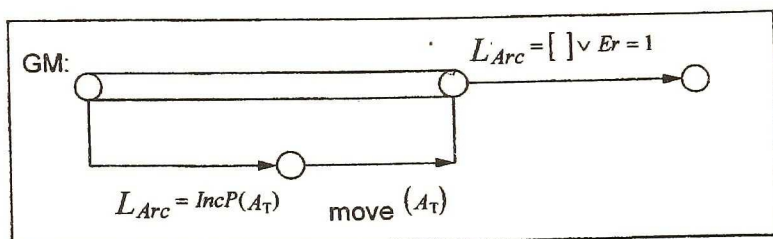


Рис. 1.6

Граф-машина на каждой итерации формирует список дуг L_{Arc} , исходящих из текущей вершины A_T . Если список L_{Arc} не является пустым, то модуль $move(A_T)$ проверяет предикаты $P_{i_0}, P_{i_1}, \dots, P_{i_L}$, помечающие дуги. Проверки производятся в соответствии с приоритетами дуг. Для первого встретившегося истинного предиката P_{i_k} модулем $move(A_T)$ вызывается к исполнению соответствующая вершина A_{i_k} , после чего статус текущей вершины A_T переназначается на вершину A_{i_k} . Работа алгоритма продолжается, пока не возникает ситуация окончания работы: $L_{Arc} = []$ или $Er=1$.

В целях отладки граф-программ агрегатов или измерения динамических характеристик агрегатов в ГСП применяются модифицированные версии граф-машины GM. Для этого достаточно подвергнуть незначительным изменениям механизм активизации модулей объектов, из которых составлена граф-программа. Например, на рис. 1.7 представлена одна из возможных модификаций механизма вызова объекта A_{i_k} .

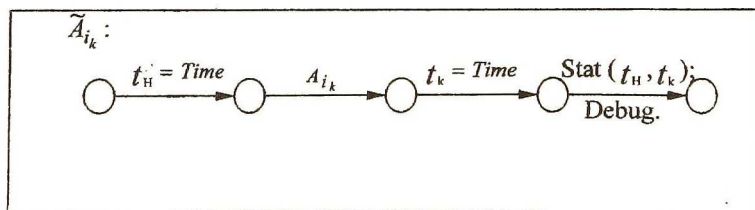


Рис. 1.7

Суть изменений заключается в том, что перед вызовом объекта A_{i_k} и после завершения его работы фиксируются моменты времени начала t_H и конца t_k работы объекта A_{i_k} . Информация передается в модуль обработки статистических данных $Stat(t_H, t_k)$. Каждый раз при этом вызывается модуль-отладчик $Debug$, который организует взаимодействие разработчика программы

с GM в процессе отладки программы, а также служит целям визуализации процессов отладки.

1.5 Модель межмодульного информационного интерфейса технологии ГСП

Проблема передачи информации от одной программы к другой традиционно представляет собой одну из наименее популярных проблем в среде программистов и одну из проблем, которая служит источником наибольшего количества ошибок в разрабатываемом программном обеспечении. В данной проблеме следует выделить четыре аспекта:

1). Проблема, связанная с реализацией *механизма доступа* подпрограмм к необходимой информации. Суть ее сводится к выяснению воздействий, производимых на передаваемые подпрограмме фактические параметры и возможной модификации значений соответствующих формальных параметров в теле подпрограммы.

2). Проблемы *распределения и управления* памятью ЭВМ.

3). Проблема *отладки* межмодульного информационного интерфейса, которая связана с декларативным стилем его построения, когда операторы объявления типов данных, самих данных, выделение оперативной памяти под данные, а также указание областей их действия, “размазаны” по тексту программы. В таких условиях идентификация ошибок в информационном интерфейсе превращается в нетривиальную задачу.

4). Проблема *массовости* используемых данных. Современное программное обеспечение характеризуется большой сложностью и значительными размерами. Количество данных, используемых программой, нарастает комбинаторными темпами. Естественно, что большие объемы циркулирующей в программах информации ставят новые задачи по их управлению.

Анализ ситуации показывает, что значительная часть проблем обусловлена слишком большими возможностями, предоставляемыми в языках программирования для построения межмодульных интерфейсов. Механизмы локального и глобального описания данных, динамическое распределение памяти, передача параметров по адресу и значению - оправданы, когда ресурсы ЭВМ были сильно ограничены. В настоящее время полный спектр таких возможностей требуется при написании незначительного количества узкоспециализированных программ. Подавляющее большинство программ, в условиях практически неограниченных ресурсов ЭВМ, может обойтись более скромными возможностями.

Из сказанного следует, что решение обозначенных проблем лежит в плоскости систематизации и введения разумных ограничений на способы и методы построения межмодульного информационного интерфейса, а также за счет разработки средств автоматизации построения таких интерфейсов.

Так, например, на языке FOXPRO обмен информацией между подпрограммами обеспечивается за счет реализации механизма глобального описания переменных, если не учитывать возможность обмена данными через таблицы базы данных. На языке ПРОЛОГ проблема организации информационного интерфейса вообще отсутствует, поскольку в концепциях языка заложены идеи логического вывода целевых условий в соответствии с аксиоматикой предметной области, а не выполнения процедурных действий над данными. При этом все операции, выполняемые над данными, скрыты (осуществляются автоматически) от пользователя. Вероятно, благодаря именно этой особенности, язык ПРОЛОГ позволяет порождать на редкость надежные программные коды в очень короткие сроки.

В языках, ориентированных на объекты, новая парадигма программирования невольно отделила описание структур данных (классов) от подпрограмм, их использующих (методов). В новых концепциях основное внимание программиста сосредоточено на формировании модели данных описываемого объекта, чем обеспечивается более высокий уровень надежности разрабатываемых программ. Не менее важен факт автоматического порождения поколений данных при создании новых объектов. Однако дела обстоят благополучно до тех пор, пока построенная "пирамида" классов удовлетворяет поставленным перед программистом целям. Любые даже незначительные изменения целевых установок на разработку программного обеспечения или в концепциях построения модели структуры данных требуют серьезного изучения используемой иерархии классов, их свойств и методов и, в конечном итоге, значительных усилий по модификации программного обеспечения.

В технологии ГСП также вводится стандарт на организацию межмодульного информационного интерфейса. Стандарт обеспечивается выполнением пяти основных правил:

- 1). Вводится единое для всей предметной области хранилище данных, актуальных для ПОП. Полное описание данных размещено в *словаре данных* ПОП. Любые переменные, не описанные в *словаре данных*, считаются локальными данными тех объектов ГСП, где они используются.

- 2). В пределах ГСП описание типов данных размещается централизованно в *архиве типов данных*.

- 3). В базовых модулях в качестве механизма доступа к данным допускается только передача параметров *по адресам* данных.

- 4). Привязка данных объектов ПОП реализована в паспортах объектов ПОП.

- 5). В технологии ГСП не рекомендуется использовать иные способы организации межпрограммных связей по данным.

Предложенный стандарт позволяет полностью отделить задачу построения межмодульного информационного интерфейса от кодирования

процедурной части программы, а также частично автоматизировать процессы построения информационного интерфейса.

С информационной точки зрения каждый объект ГСП f_i представляет собой функциональное отображение области определения объекта D_i^{in} на область значений D_i^{out} :

$$f_i: D_i^{in} \rightarrow D_i^{out}.$$

В общем случае $D_i^{in} \cap D_i^{out} \neq \emptyset$ (в объекте могут быть модифицируемые данные) и $D_i^{in}, D_i^{out} \in D$, где D полная область данных ПОП. Для двух произвольных объектов ПОП f_i и f_j в общем случае справедливо: $(D_i^{in} \cup D_i^{out}) \cap (D_j^{in} \cup D_j^{out}) \neq \emptyset$.

Формально сущность проблемы организации передачи данных между объектами в рамках некоторого модуля-агрегата f_Σ можно определить как задачу построения области данных агрегата f_Σ - $D_\Sigma = D_\Sigma^{in} \cup D_\Sigma^{out}$ и установления соответствий между данными $D_\Sigma = \{d_1, d_2, \dots, d_{n_\Sigma}\}$ и данными $D_i = \{d_1^i, d_2^i, \dots, d_{n_i}^i\}$ объектов $f_1, f_2, \dots, f_m$, из которых составлен агрегат f_Σ (см. рис.1.8).

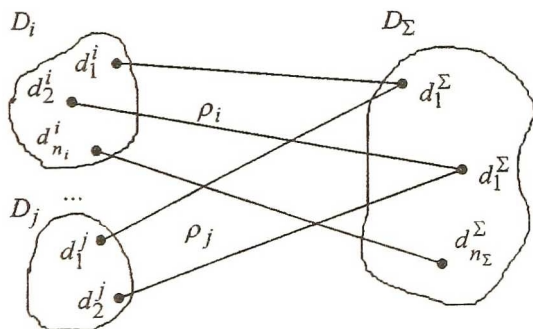


Рис 1.8 Информационный межмодульный интерфейс

В традиционном программировании любое, даже незначительное изменение структур данных в модулях вызывает необходимость "ручной" переделки соответствующих информационных связей ρ_j . В технологии ГСП этот процесс автоматизирован за счет использования отношений ρ_j , описанных и хранящихся отдельно от программной реализации объекта в собственном паспорте.

1.5.1. Информационный интерфейс базовых модулей

Как уже говорилось, первоначальным строительным материалом (локальными вычислимыми функциями) в ГСП являются базовые модули. Информационный интерфейс базовых модулей представляет собой некое отношение, устанавливающее связь между типами данных базового модуля (формальными параметрами) и данными предметной области. В результате установления связи порождаются новые объекты либо акторы, либо предикаты.

Отношение ρ_j в ГСП формируется “паспортизацией” типов данных базовых модулей, т.е. за счет “опредмечивания” формальных параметров базовых модулей. В этом смысле отношение ρ_j является по сути “паспортом” модуля и вместе с базовым модулем определяют понятие актора или предиката (см. рис. 1.9).

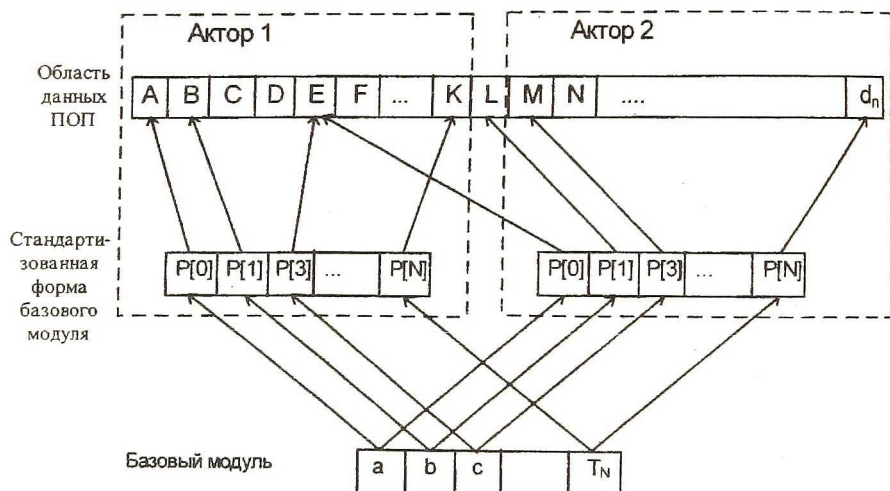


Рис. 1.9. Межмодульный интерфейс базового модуля

Таким образом, каждый актор (предикат) через массив указателей $P[N]$ “знает”, над какими данными предметной области необходимо производить функциональные преобразования, описанные в соответствующем базовом модуле.

1.5.2 Информационный интерфейс агрегатов

Информационный интерфейс объектов-агрегатов в технологии ГСП строится автоматически на основе паспортов объектов, из которых “сконструирован” сам агрегат. Однако в случае агрегата его массив указателей

$P[M]$ ссылается не на данные ПОП, а на массивы указателей $P_j[N_j]$ объектов, из которых составлен агрегат (актор, предикат или другой агрегат).

Например, для агрегата, представленного на рисунке 1.10, его массив адресов содержит ссылки на массивы адресов объектов A_1, A_2, G_1, P_1, P_2 , т.е. устанавливается соответствие вида:

$$P_{G_2}[] = \{addr(P_{A_1}[]), addr(P_{A_2}[]), addr(P_{G_1}[]), addr(P_{P_1}[]), addr(P_{P_2}[])\},$$

где $addr()$ - функция "взятия" адреса структуры данных (в языке Си для этого используется операция - "&").

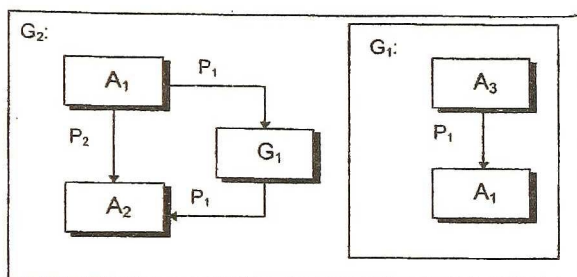


Рис. 1.10.

В полном виде модель информационного интерфейса агрегата G_2 показана на рисунке 1.11. Из рисунка видно, что непосредственно с данными ПОП оперируют только акторы и предикаты агрегата, остальные объекты агрегата работают с данными ПОП опосредованно с помощью механизма косвенной адресации. В этом смысле модель межмодульного информационного интерфейса агрегата представляет собой иерархическую древовидную структуру.

При формировании нового агрегата, использующего в качестве объекта агрегат G_2 , древовидная структура "старого" агрегата как поддерево войдет в структуру нового агрегата. При этом новый агрегат будет "наследовать" информационные связи агрегата G_2 .

Структура автоматически компилируемого текста объекта-агрегата достаточно проста. Текст программы состоит из: стандартного для всех объектов заголовка; структур данных, описывающих граф-программы; и обращения к стандартной программе граф-машине. С помощью компилятора языка Си текст агрегата компилируется в объектный модуль и помещается в библиотеку объектных модулей ПОП.

Особенностью предложенного способа построения межмодульного информационного интерфейса является то, что формируемые (автоматически или автоматизировано) программные коды и информационные связи “пространственно” не зависят друг от друга. Модификация любого из объектов (актора, предиката или агрегата) не требует переделки кодов других объектов,

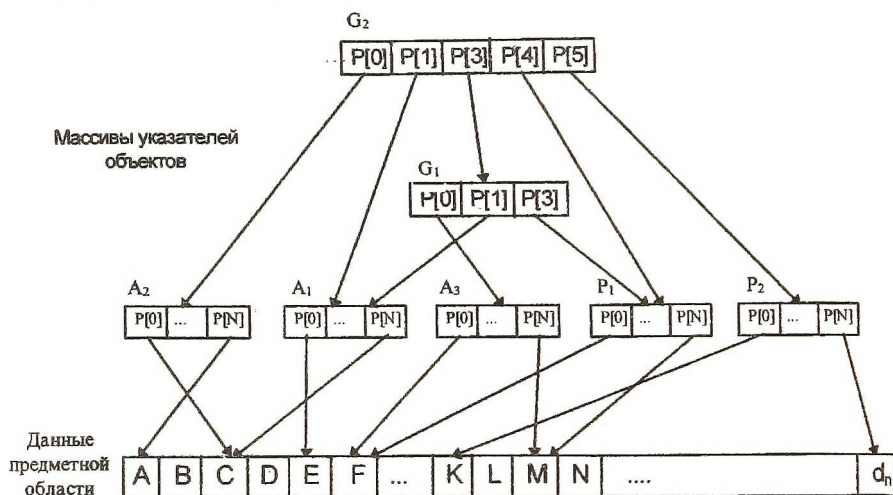


Рис. 1.11. Межмодульный интерфейс агрегата G_2

входящих в ПОП. Более того, на самом деле при порождении акторов, предикатов или агрегатов никаких программных конструкций, описывающих информационный интерфейс, для них не создаются. Акторы и предикаты это лишь “фантомные” конструкции, не имеющие текстов программ, а описанные в виде “паспортов” в информационном фонде ПОП. Информационные связи агрегатов косвенно содержатся в структуре граф-программы. Программная конструкция, описывающая информационный интерфейс, автоматически порождается при компиляции текста “исполняемой” программы любого объекта ПОП - EXE-модуля.

1.5.3. Информационный интерфейс EXE-модулей

Межмодульный информационный интерфейс (МИИ) в виде кодов программы генерируется всякий раз, когда на основе объекта (актора или агрегата) порождается соответствующий исполняемый программный модуль (EXE-модуль).

Исполняемый аналог объекта представляет собой достаточно простую конструкцию, состоящую из МИИ и вызова соответствующего объектного модуля объекта ПОП.

Программа, описывающая МИИ, состоит из двух частей:

1) операторов объявления структур данных объекта, для которого строится EXE-модуль, при этом данным выделяется оперативная память в соответствии с описаниями, хранящимися в словаре данных ПОП и архиве типов данных,

2) древовидной системы массивов указателей с инициализацией их значений.

Например, МИИ агрегата G_2 может быть описан так, как это представлено на рисунке 1.12. Общая область данных агрегата формируется в результате объединения всех данных, используемых в акторах и предикатах граф-программы: $D_\Sigma = UD_j$, где D_j - данные, описанные в паспортах акторов и предикатах. Агрегаты в формировании D_Σ не участвуют, поскольку они осуществляют только транзитную передачу информации посредством механизма косвенной адресации до исполнимых модулей акторов или предикатов.

```
...
static double A=0;
static double B=1;      /* Формирование области данных агрегата G2 */
static double C=10;     /* Выделение памяти под данные агрегата с
                        /* инициализацией начальных значений */

...
static double dn;
//
static void *A1[ ]={&E,...,&C};
static void *A2[ ]={&C,...,&A}; /* Дерево информационных связей */
static void *A3[ ]={&F,...,&M};
static void *P1[ ]={&F,...,&M};
static void *P2[ ]={&K,...,&dn};
static void *G1[ ]={A3,A1,P1};
static void *G2[ ]={A2,A1,G1,P1,P2};
```

Рис. 1.12 Автоматически порождаемый текст МИИ агрегата G_2

Дерево информационных связей порождает специальный двухпроходный алгоритм, занимающийся, главным образом, упорядочиванием размещения массивов указателей объектов в соответствии со структурной иерархией, используемых в агрегате объектов.

Таким образом, межмодульный информационный интерфейс в процессе разработки объектов ГСП строится неявно, размещаясь в информационных структурах фонда ГСП (словаре данных ПОП, архиве типов данных, паспортах акторов и предикатов, а также описаниях структуры агрегатов). Формирование МИИ происходит на этапе построения EXE-модуля любого объекта ПОП. Такое положение дел в технологии ГСП позволяет полностью отделить процессы описания процедурной части алгоритма от описания межмодульного информационного интерфейса и автоматизировать порождение последнего.

Глава 2.

КОНСТРУИРОВАНИЕ ОБЪЕКТОВ ТЕХНОЛОГИИ ГСП

Первоначально система GRAPH разрабатывалась как средство автоматизации программирования, ориентированное на создание программного обеспечения САПР технических изделий. Автоматизация программирования предполагает, в первую очередь, *сокращение сроков разработки программных продуктов и повышение их качественных характеристик*. Существует два способа достижения этой цели.

Первый способ связан с автоматическим синтезом программ в соответствии с совокупностью так или иначе представленных *знаний* о предметной области. Автоматически построенные программы заведомо правильны и, как правило, не требуют отладки. При этом резко возрастает производительность труда программиста. К системам автоматизации программирования можно отнести, в первую очередь, языки логического программирования (ПРОЛОГ, LISP, ПРЛЗ), объектно-ориентированное программирование (C++, Паскаль), CASE-технологии (ISaGRAF) и т.д.

Однако автоматический синтез программ во всех областях применения компьютеров пока невозможен. Более того, он подходит лишь в достаточно простых, и хорошо изученных областях, для которых легко определить и описать базу знаний. К сожалению, такая ситуация встречается не часто. Поэтому подобные системы синтеза программ прежде всего хорошо зарекомендовали в классических, давно устоявшихся разделах математики, электроники, где все законы, правила и формулы давно известны - они достаточно просты и довольно полно отражают предметную область.

Второй способ автоматизации программирования ориентирован на приближение языка программирования к образному способу мышления человека, что выражается в бездирективном методе организации разработки программ. Повышение производительности труда программиста в этом случае связывают с большей наглядностью (понятностью) программ и более комфортными условиями труда, что, в конечном итоге, приводит к повышению надежности результирующих программных продуктов.

Неслучайно, что в настоящее время появилось большое количество визуальных средств программирования. К ним относятся разнообразные средства разработки оконных интерфейсов, "визарды" в языках управления базами данных, графические инструментальные средства в CASE-системах, многочисленные проблемно и предметно ориентированные языки программирования.

В то же время, визуальное программирование расширяет "армию труда" в сфере программирования, поскольку к узкому кругу профессиональных программистов, в новых условиях, подключается большое количество

“любителей”, способных разрабатывать качественные программные продукты на новых средствах программирования.

В технологии ГСП используются оба способа автоматизации программирования. С одной стороны, технология ГСП применяет визуальный способ кодирования программ, с другой стороны - автоматизирован синтез многих компонент разрабатываемого программного обеспечения. Например, автоматически синтезируются коды агрегатов, in-line акторы, межмодульный информационный интерфейс. Кроме того, технология ГСП имеет интеллектуальную поддержку, обеспечивающую разработку надежных программных модулей.

Визуальное программирование в ГСП возможно на непустом множестве объектов ПОП. Объекты в ГСП порождаются либо традиционным “ручным” способом, при кодировании на базовом языке программирования базовых модулей, либо автоматизированным при синтезе акторов, агрегатов, предикатов, модулей типа in-line, а также в результате инкапсуляции агрегатов. Доля неавтоматизированных модулей в ГСП по мере развития ПОП постоянно уменьшается, а степень автоматизированности программирования увеличивается.

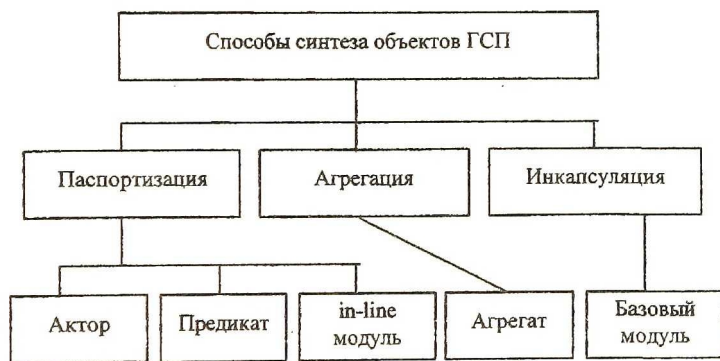


Рис 2.1. Способы синтеза объектов ГСП

Можно выделить три способа автоматического синтеза объектов ГСП (см рис 2.1): паспорттизацию, агрегацию и инкапсуляцию

Паспорттизация из базовых модулей (типов объектов) порождает полиморфные объекты - акторы или предикаты, т.е. одна и та же синтаксическая конструкция (базовый модуль) порождает различные семантические формы объектов.

Агрегация из совокупности имеющихся семантических конструкций порождает новую семантическую конструкцию.

Инкапсуляция реализует, в определенном смысле, обратное двум предыдущим действие: порождает синтаксическую конструкцию (базовый модуль) из выбранной семантической формы объекта.

В данной главе рассматриваются вопросы конструирования объектов на концептуальной основе технологии графо-символического программирования, а также проблема классификации данных агрегатов ПОП.

2.1. Конструирование объектов инкапсуляцией агрегатов

Автоматизированное программирование в ГСП возможно на основе множества объектов. Значительная часть из них в качестве реализации типа объекта имеет базовые модули, которые формируются “ручным” способом - программированием на базовом языке. В принципе, эта ситуация не столь критична, поскольку в большинстве случаев в качестве базовых модулей можно использовать подпрограммы из обширных библиотек стандартных программ языка программирования, практически не занимаясь “ручным” кодированием программ. Тем не менее, проблема автоматизации порождения базовых модулей представляет несомненный практический интерес. В качестве решения этой проблемы предлагается использовать операцию *инкапсуляции* агрегатов ГСП.

Понятие инкапсуляции возникло в “недрах” объектно-ориентированного программирования, как возможность “скрывать” частности от целого. Такой подход позволяет программисту не знать частных деталей реализации программной системы, осуществлять конструирование из элементов, реализация которых скрыта от него под оболочкой модуля. По существу разговор идет о механизме сокрытия внутренних переменных класса и описания интерфейса класса.

Соккрытие внутренних переменных объекта актуально и для ГСП. Например, в процессе построения такого сложного объекта, как термогазодинамическая модель авиационного двигателя, в процессе агрегации объекта из объектов более низкого иерархического уровня (узлов ГТД) в последнем накапливается большое количество данных, используемых в моделях узлов двигателя. В среднем, число параметров для модели ГТД колеблется от нескольких сотен до нескольких тысяч. Большинство параметров агрегата являются внутренними и используются для передачи информации от одного объекта к другому. При построении объекта более высокого иерархического уровня, например, летательного аппарата, большинство параметров двигателя в рамках нового объекта практически не нужны. В этом случае помогает механизм инкапсуляции внутренних переменных модели ГТД.

Инкапсуляция агрегатов в ГСП реализуется следующим образом:

- 1). На первом этапе по паспортам акторов и предикатов формируется область данных агрегата.

2). На втором этапе данные агрегата разбиваются на четыре класса: исходные данные, выходные данные, модифицируемые данные и внутренние данные. Здесь под внутренними данными понимаются данные, которые идентифицируются по признаку “выход предшествующего в схеме управления объекта - вход в любой из объектов, следующих за предшествующим”.

3). Паспортные данные агрегата предъявляются разработчику, и он, с помощью построенной классификации данных, все множество данных разбивает на две группы: внутренние (инкапсулированные) данные и внешние данные, специфицирующие тип нового базового модуля.

4). На последнем этапе автоматически синтезируется текст нового базового модуля.

Синтез текста нового базового модуля реализуется за счет построения заголовка программы, в который включается список всех типов данных, описанных разработчиком как внешние в том порядке, как он указывает. Для внутренних данных строится межмодульный информационный интерфейс также, как это делается для исполнимых модулей.

Рассмотрим простую задачу построения базового модуля инкапсулированием агрегата A_{poz} , реализующего позиционирование некоторого слова b в строке символов a (см. рис.2.2). Для этих целей используется базовый модуль Poz , составленный на языке Си, с использованием стандартной подпрограммы $strcmp()$.

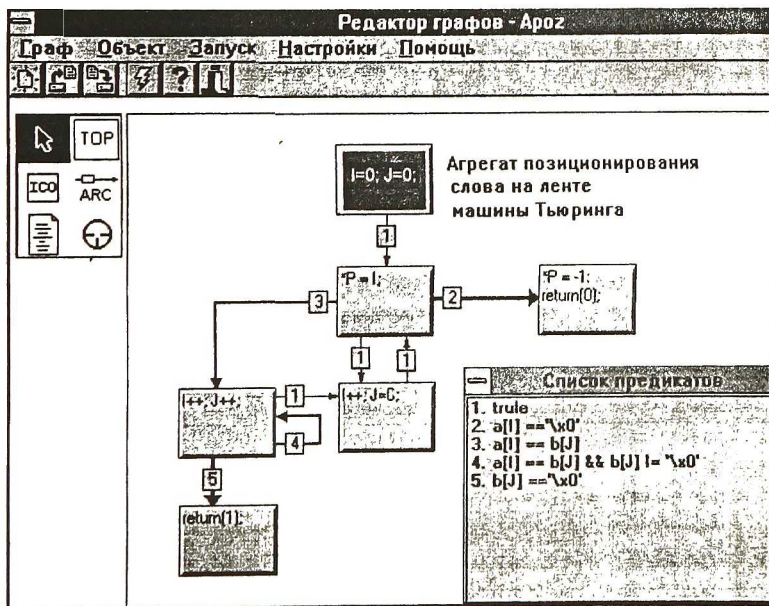


Рис. 2.2. Агрегат позиционирования слова b на ленте a .

рисунке 2.4 представлен текст аналогичного базового модуля, закодированного на языке Си.

```
/*Позиционирование строки В в строке А*/
#include <string.h>
#include <stdio.h>
#include "C:\graph\grfpolh\MACH.H"
int Poz(TAPE a,WORD b,int *p)
{
    int I,J;
    I=0; J=0;
    m1::;
    *p=I;
    if (a[I] == '\x0') {*p=-1; return(0);}
    m2::;
    if (a[I] == b[J])
        { I++; J++;
          if (b[J]!='\x0') goto m2;
          else return(1);
        }
    else
        {I++; J=0;goto m1;};
    return(1);
}
```

Рис.2.4. Текст базового модуля Poz, составленного на языке Си

Сравнение текстов одной и той же программы, составленных средствами системы GRAPH (рис. 2.2) и на языке Си, демонстрирует выразительные достоинства граф-программ.

2.2 Использование рекурсии для конструирования объектов ГСП

Рекурсия является мощным методом программирования. Рекурсия, как возможность ввести в определение объекта ссылку на сам объект, часто возникает в программировании. Рекурсивное описание имеют многие математические объекты, например, функция $n!$, полиномы Чебышева, арифметические и геометрические прогрессии и т.п. Рекурсивные определения удобны при обработке сложных структур данных. В языке ПРОЛОГ рекурсия приобретает особую важность, поскольку позволяет моделировать циклические конструкции типа `while...do...`.

С точки зрения теории программирования рекурсивные алгоритмы интересны тем, что являются практическим воплощением такой математической конструкции как *рекурсивные функции*.

Появление в математике частично рекурсивных функций связано с решением проблемы точной формулировки понятия *алгоритма*. Исследование этого вопроса в 30-х годах нашего века привело к созданию теории рекурсивных функций.

Класс частично рекурсивных функций описывает специфические функции от конечного числа натуральных аргументов с натуральными значениями. Эти функции обладают некоторыми эффективными процедурами нахождения значения функции (если оно существует) по заданным значениям аргументов. Впервые класс частично рекурсивных функций определили Ж. Эрбан, К. Гедель, С. Клини [57]. В соответствии с тезисом Черча только частично рекурсивные функции являются эффективно (алгоритмически) вычислимыми на ЦВМ.

С другой стороны, класс эффективно вычисляемых функций не ограничивается множеством натуральных чисел. Существует множество алгоритмов, оперирующих на более сложных конструктивных областях: на множестве слов алфавита, формул некоторой теории, графов и т.д. В то же время, теория нумераций позволяет свести любые эффективно вычисляемые функции к соответствующим частично рекурсивным, подобно тому, как происходит машинный перевод текста программы с одного языка программирования на другой.

Рекурсивные конструкции играют значительную роль и в технологии ГСП. Основу таких конструкций составляют *рекурсивные граф-агрегаты*.

Под *рекурсивным граф-агрегатом* будем понимать агрегат, в котором одна или несколько вершин помечены именем самого агрегата.

С практической точки зрения рекурсивные агрегаты позволяют не только программировать рекурсивные функции, но и порождать графы с динамически изменяемой структурой. Такие динамические конструкции часто требуются при программировании, например, в области искусственного интеллекта.

Рассмотрим особенности построения рекурсивных конструкций. В математике рекурсивные функции определяются следующим образом [54]:

$$\begin{aligned} f(x_1, \dots, x_n, 0) &= g(x_1, \dots, x_n); \\ f(x_1, \dots, x_n, y+1) &= h(x_1, \dots, x_n, y, f(x_1, \dots, x_n, y)), \end{aligned} \quad (2.1)$$

где f рекурсивная функция, g n -местная функция, h $(n+2)$ -местная функция. Первое равенство в (2.1) описывает "граничное (начальное) условие" рекурсии, второе - вводит рекуррентное соотношение для вычисления функции f .

Рекурсивные алгоритмы практически полностью повторяют формулу (2.1). Любая рекурсивная программа должна включать, по крайней мере, по одной из перечисленных ниже компонент:

1) Не рекурсивную фразу, определяющую *исходный* или *конечный* вид компоненты программы, которая позволяла бы вычислять функцию *непосредственно*, без рекурсивного вызова (первая формула (2.1)).

2) Рекурсивный вызов самой себя из тела программы (вторая формула системы (2.1)).

Для того, чтобы выполнение рекурсивной программы могло быть завершено, необходимо, чтобы существовала некоторая связанная с этой программой *управляющая величина* - целое неотрицательное число, относительно которой можно было бы с уверенностью сказать, что она строго убывает (возрастает) при каждом рекурсивном вызове. В определении (2.1) такой величиной является переменная y . Переменные $x_1, \dots, x_n$ соответствуют формальным параметрам рекурсивной программы.

Однако техника практической реализации рекурсии в языках программирования несколько отличается от формального описания с помощью рекурсивных функций. Трансляторы языка реализуют выполнение рекурсии в два прохода. "Прямой" ход связан с вычислениями функции от *исходного значения* до *граничного условия*, при этом в динамически распределяемый стек памяти вычислительной машины запоминаются все вызовы программы вместе со значениями ее параметров, полученных на каждом шаге работы программы. "Обратный" ход восстанавливает состояние программы в порядке обратном прямому ходу. С помощью рекурсивных функций такое положение дел можно описать с помощью системы рекурсивных функций:

$$\begin{cases} f_1(x_1, \dots, x_n, 0) = g(x_1, \dots, x_n); \\ f_1(x_1, \dots, x_n, y - 1) = h_1(x_1, \dots, x_n, y, f_1(x_1, \dots, x_n, y)), \\ f_1(x_1, \dots, x_n, y + 1) = h_2(x_1, \dots, x_n, y, f_1(x_1, \dots, x_n, y)), \end{cases} \quad (2.2)$$

где h_1 и h_2 - реализации прямого и обратного хода рекурсии.

В технологии ГСП принята "обобществленная" модель обмена данными между объектами ПОП в соответствии с принятой концепцией организации межмодульного информационного интерфейса (см. пп. 1.5). Последнее означает, что рекурсивные функции не имеют формальных параметров, что существенно сокращает размеры динамически размещаемого стека, упрощает формальную модель (2.2) и несколько усложняет в отдельных случаях составление рекурсивных агрегатов.

Таким образом, в качестве формальной модели рекурсивного агрегата можно рассматривать следующую модель:

$$\begin{cases} f_1(0) = c; \\ f_1(y - 1) = h_1(y, f_1(y)), \\ f_1(y + 1) = h_2(y, f_1(y)), \end{cases} \quad (2.3)$$

причем управляющий параметр y на самом деле "скрыт" от пользователя стандартным механизмом организации рекурсии, принятом в языке программирования.

Модель (2.3) подсказывает правила построения рекурсивных агрегатов.

Например, граничное условие не может быть корневой или “внутренней” вершиной графа, поскольку в соответствии с моделью (2.3) граничное условие является последним вычислением рекурсивной функции на прямом ходе рекурсии. Здесь под “внутренними” вершинами понимаются вершины, имеющие входящие и выходящие дуги. Тогда граничные условия должны топологически располагаться только в терминальных вершинах графа.

“Рекурсивные” вершины не могут располагаться в корневой вершине графа, поскольку в этом случае отсекается доступ к граничным вершинам, и рекурсия становится бесконечной. В то же время, рекурсивные объекты могут располагаться во внутренних или концевых вершинах графа.

Наличие “граничной” вершины является обязательным условием для рекурсивных агрегатов.

Рекурсивный агрегат, удовлетворяющий перечисленным выше правилам, будем называть *правильно построенным*.

Используя эти правила, нетрудно конструировать рекурсивные агрегаты. Например, на рисунке 2.5 представлен рекурсивный агрегат FACTORIAL, описывающий функцию $n!$.

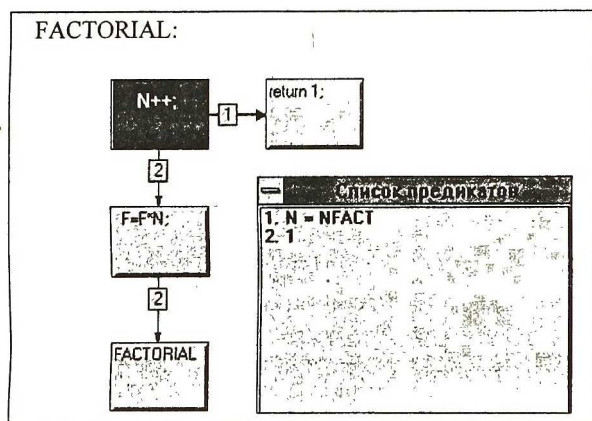


Рис. 2.5. Рекурсивный агрегат функции $n!$

В примере данное NFACT задает значение аргумента функции, F - вычисляемое значение факториала. В данном случае специальных граничных условий не требуется. Достаточно проверки условия завершения рекурсивных циклов ($N == NFACT$), поэтому в качестве граничного условия используется фиктивная вершина RETURN(1).

В данном примере реализована простая прямая рекурсия. Конечный результат (вычисление функции $n!$) достигается на прямом ходе рекурсивного алгоритма. Более тонкие алгоритмы используют и обратный ход рекурсии.

На рисунке 2.6 представлена рекурсивная структура объекта, который может служить основой для разработки алгоритма решения задачи о ханойских башнях. В вершине 1 декрементируется число дисков на очередном шаге прямого хода рекурсивного алгоритма. Вершины 2 и 2' реализуют рекурсивное обращение к агрегату graf (решения задачи о ханойских башнях). Вершина 4 описывает тривиальное граничное условие `return(1)`. Вершина 5 "запоминает" число дисков в точке ветвления алгоритма и устанавливает флаг F в режим "перезапуска" алгоритма по новой ветке. Вершина 3 восстанавливает число дисков при возврате в точку ветвления.

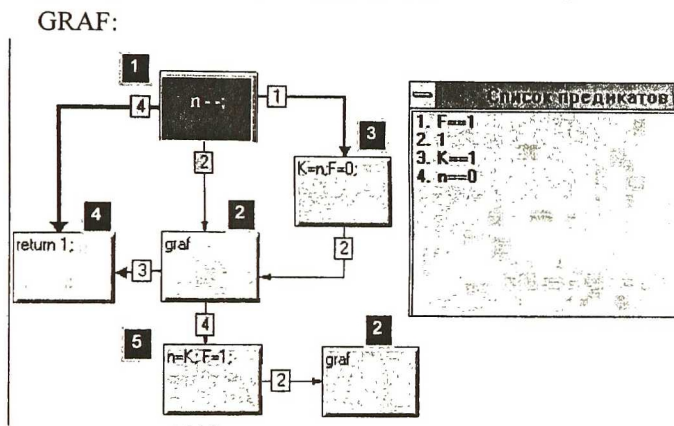


Рис. 2.6. Разветвленный рекурсивный агрегат

При выбранной схеме решения головоломки вместо прямой рекурсии следовало бы применять косвенную рекурсию, когда в вершинах 2 и 2' вызывался бы агрегат `grafM`, имеющий обращение к агрегату `graf` и реализующий перенос дисков со стержня на стержень. Однако, в данном случае нас более интересуют возможности рекурсивных агрегатов порождать динамически изменяющиеся графы.

Так, например, при $n=3$, нерекурсивный аналог агрегата `graf` будет выглядеть так, как это представлено на рисунке 2.7. Для других значений n граф будет выглядеть уже совершенно иначе.

Применение технологии ГСП для разработки рекурсивных алгоритмов удобно тем, что при их отладке в автоматическом режиме можно порождать структуры, подобные той, которая представлена на рис 2.7, и при различных значениях исходных данных "видеть" развитие вычислительного процесса.

Применение рекурсии, основанное на использовании "штатных" средств, предоставляемых транслятором языка, чревато неприятными последствиями, связанными с непредсказуемым переполнением динамически изменяемого

стека программы. Последнее можно избежать, если реализацию рекурсивных вызовов оформить автономно от системных средств. В технологии ГСП такое решение находится достаточно просто.

Область данных рекурсивного агрегата расширяется стековой структурой данных (списком) L_R , предназначенной для хранения списка, рекурсивно вызываемых вершин графа на прямом ходе рекурсии. В “голове” стека всегда находится последняя из рекурсивно вызываемых вершин графа. В стек записываются номера рекурсивных вершин в нотации, принятой для данного агрегата (вершины всех агрегатов имеют сквозную внутреннюю нумерацию).

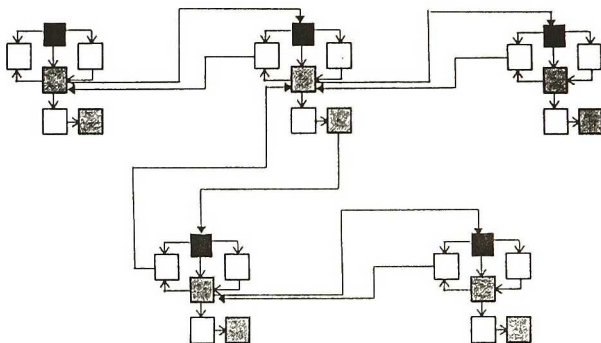


Рис. 2.7.

Рекурсивные вершины графа пометим индексом R . Пусть $A_{i_k}^R$ рекурсивные вершины графа, у которых имя объекта совпадает с именем агрегата, но топологическое расположение которых на графе достаточно произвольно. Через A_0 обозначим корневую вершину графа.

Введем флаг F , устанавливающий направление развития вычислительного процесса рекурсивного алгоритма ($F=1$ – “прямой” ход рекурсии, $F=0$ – обратный ход рекурсии).

Далее нам потребуются две функции манипулирования со стеком L_R :

$\text{App}(L_R, A_{i_k}^R)$ – добавление в голову списка рекурсивной вершины $A_{i_k}^R$;

$\text{Read}(L_R, A_{i_k}^R)$ – чтение и удаление из головы списка рекурсивной вершины $A_{i_k}^R$.

Введем изменения в структуру алгоритма граф-машины GM , управляющей развитием вычислительного процесса на агрегате. Далее используется система обозначений, принятая в разделе 1.4.

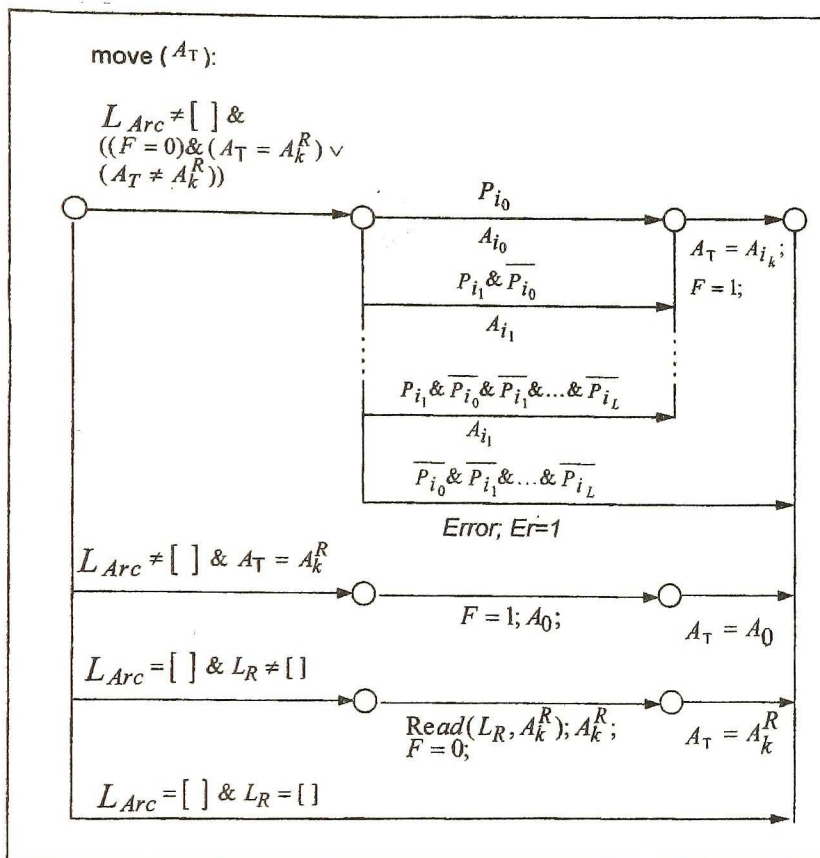


Рис. 2.8

На рисунке 2.8 представлена модифицированная схема алгоритма планирования перехода агрегата в следующее состояние. В отличие от алгоритма интеррогативного управления (см. рис. 1.5) данная схема имеет следующие отличия:

- 1). По “старой” схеме вычислительный процесс развивается, если текущая вершина не является тупиковой (концевой). Переход из рекурсивной вершины возможен только на обратном ходе рекурсии ($F=0$).
- 2). Если реализуется переход из рекурсивной вершины на прямом ходе рекурсии, то флаг F устанавливается в 1, а управление передается в корневую вершину графа A_0 .
- 3). В случае, если из вершины нет исходящих дуг (список L_{Arc} пуст) и стек L_R не пуст, то из головы списка выбирается вершина возврата рекурсии и

именно этой вершине передается управление. При этом из списка удаляется “отработанный” рекурсивный вызов. Флаг F переводится в состояние обратного хода рекурсии.

4). Работа алгоритма заканчивается, если достигнута тупиковая вершина графа и стек рекурсивных вершин пуст.

Механизм активации рекурсивных объектов реализуется в соответствии со схемой, представленной на рисунке 2.9. На прямом ходе рекурсии в список L_R добавляется еще одна вершина. На обратном ходе никакие действия не производятся.

Представленный алгоритм, по существу, моделирует работу транслятора. Однако, в данном случае учитываются особенности организации межмодульного информационного интерфейса, принятого в технологии ГСП и в стек не заносится состояние памяти рекурсивно вызываемого объекта, что уменьшает размеры стека. Кроме того, управление стеком теперь находится “в руках” граф-машины, что позволяет держать под контролем состояние памяти вычислительной машины и при необходимости подключать дополнительные ресурсы, например, память на жестких дисках.

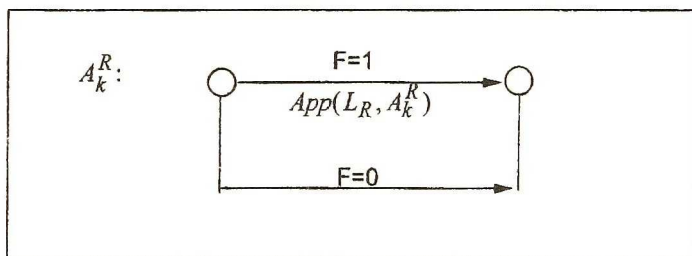


Рис. 2.9

В отдельных случаях “обобществленная” модель организации памяти агрегата при разработке рекурсивных алгоритмов вызывает некоторые неудобства. Часто значительно удобнее вместе с рекурсивными вызовами запоминать конкретизации некоторых параметров агрегата в месте ветвления алгоритма. Ситуацию можно исправить, если ввести локальные для агрегата данные, настроенные на общую область данных, и вместе со стеком L_R собирать в отдельном стеке поколения конкретизаций значений локальных данных.

Рекурсивные программы, как правило, внешне просты, но содержательно очень сложны. У многих программистов наблюдается отрицательное отношение к рекурсии, потому что она требует порой высокого уровня абстракции. Последнее, по нашему мнению, связано с практически полным отсутствием визуальных средств представления рекурсивных алгоритмов. Более или менее полное представления о рекурсивном алгоритме возникает

только после “домысливания” всех возможных сценариев его развития в голове разработчика. По этим же причинам сложна и отладка рекурсивных алгоритмов. Рекурсивные программы не имеют конечного списка маршрутов, что затрудняет тестирование структуры программы. Исключение рекурсии позволяет применить алгоритмы структурного тестирования к нерекурсивному аналогу программы и тем самым исследовать корректность и исходной рекурсивной программы.

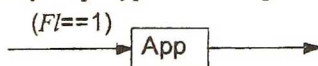
В работе [43] приводится множество примеров исключения рекурсии, основанных на использовании цикла типа “пока”. Используя возможности технологии ГСП, из любого рекурсивного агрегата можно исключить рекурсию путем превращения рекурсивного графа в “плоский” граф, не имеющий рекурсивных вершин. Поскольку данное гомоморфное преобразование производится в соответствии с жесткими правилами, то оно может быть реализовано автоматически. Построенный аналог рекурсивного агрегата для технологии ГСП имеет достаточно большое значение, поскольку позволяет реализовать процедуру тестирования рекурсивных агрегатов средствами, разработанными для обычных граф-программ. Кроме того, на “плоских” граф-программах возможно проведение мероприятий, связанных с оптимизацией ее алгоритма.

Модификация рекурсивного агрегата производится в соответствии со следующими правилами:

1. Область данных агрегата пополняется стеком, хранящим номера рекурсивных вершин и флажком Fl , определяющим направление развития рекурсии ($Fl=1$ - прямой ход рекурсии, $Fl=0$ - обратный ход рекурсии).

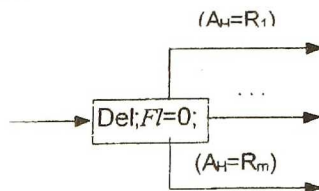
2. Рекурсивные вершины “привязываются” к “пустым” акторам, не выполняющим никаких функциональных действий (например, $\text{return}(1)$).

3. В корневую вершину от рекурсивных вершин “заводятся” звенья типа:



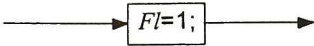
где App - автоматически порождаемый актер добавления в голову стека номера псевдо-рекурсивной вершины, ($F/=1$) - стандартный предикат проверки прямого хода развития рекурсии.

4. От концевых (граничных) вершин графа в рекурсивные вершины “заводятся” звенья типа:



где Del - стандартный актор чтения и удаления из головы стека рекурсивной вершины A_H , а также установления флага Fl в состояние возврата (обратного хода рекурсии).

5. Все дуги, входящие в рекурсивные вершины, заменяются звеньями типа:



6. Возвратные звенья, аналогичные тем, которые представлены в правиле 4, заводятся между рекурсивными вершинами.

7. Для организации выхода из рекурсии “заводится” конечная вершина END, в которую передается управление из вершин “Del;Fl=0;”, если стек пуст.

В качестве примера (см. рис.2.10) приведем результат исключения рекурсии алгоритма вычисления факториала, представленного на рисунке 2.5.

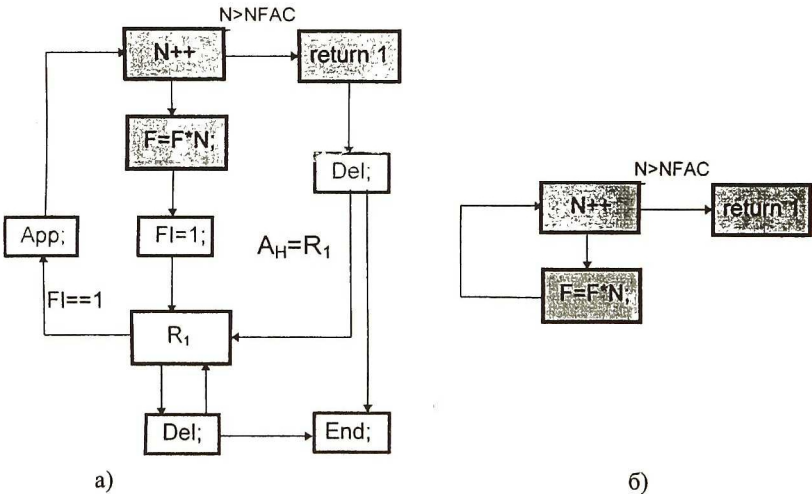


Рис. 2.10 Исключение рекурсии

В результате применения предложенного алгоритма исключения рекурсии вместо рекурсивного графа (рис. 2.5) имеем “плоский” граф, представленный на рисунке 2.10 а). Построенный граф сложнее рекурсивного аналога, но в не рекурсивном варианте к нему можно применить формальный и неформальный анализ рациональности полученной структуры. Так например, нетрудно увидеть, что все возвратные ветви графа в данном случае, когда

решение достигается на прямом ходе рекурсии, не нужны. После “оптимизации” структуры графа приходим к нерекурсивному варианту, представленному на рисунке 2.10 б), структура которого проще, чем у рекурсивного агрегата.

2.3. Классификация данных объектов ГСП

2.3.1. Проблема классификации данных агрегатов

В технологии ГСП все объекты (акторы, предикаты, агрегаты) реализуют в общем случае векторные функции многих переменных, которые можно представить в виде $D^{out} = f(D^{in})$, где D^{in} , D^{out} - множество входных и вычисляемых данных. Причем, для двух этих категорий справедливо: $D^{in}, D^{out} \in \mathcal{D}$ $D^{mod} = D^{in} \cap D^{out}$, где $\mathcal{D}$ -словарь данных предметной области. В связи с чем, к двум категориям данных *входных* и *вычисляемых*, добавляется *модифицируемый* тип данных.

Информация о разделении данных по признаку их использования на объектах необходима в следующих случаях:

1. При построении на основе объекта исполняемого EXE-модуля;
2. Для выполнения операции инкапсуляции объектов;
3. При тестировании объектов технологии ГСП.

В первых двух случаях потребность в классификации данных связана с необходимостью инициализации входных данных. Так, например, отсутствие начальных значений входных или модифицируемых переменных непременно вызовет ошибочную ситуацию при исполнении EXE-модуля. С другой стороны, при выполнении операции инкапсуляции объекта “спрятанным” данным (локальным для инкапсулированного модуля), если они входные или модифицируемые, также должно быть присвоено начальное значение.

При тестировании объектов признак классификации данных используется для выделения независимых переменных функции, в пространстве которых реализуется поиск ошибочных ситуаций.

Если для объектов типа акторов и предикатов вопрос классификации данных решается пользователем на этапе формирования паспорта модуля, то при классификации данных модуля-агрегата задача становится не столь тривиальной. В отличие от актора, в агрегате отнесение того или иного данного к определенному классу в значительной степени зависит от маршрута работы алгоритма на управляющем графе объекта.

G1 :

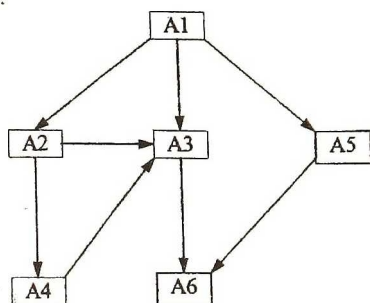


Рис. 2.11.

Пусть мы имеем агрегат G_1 , представленный на рис. 2.11. Предположим, что классификация параметров модулей A_1 , A_2 и A_3 представлена в таблице 2.1.

Таблица 2.1

Модуль	Параметр	Тип
A1	D1	входной
A2	D2	вычисляемый
A3	D2	входной

Тогда, если алгоритм будет выполняться по маршруту $A_1 \rightarrow A_3 \rightarrow A_6$, параметр D2 (из модуля A_3) должен принять значение перед началом работы программы, и его необходимо отнести к классу входных данных. Если же алгоритм пойдет по маршруту $A_1 \rightarrow A_2 \rightarrow A_3 \rightarrow A_6$, то D2 будет вычислен в модуле A_2 и, следовательно, это данное не надо инициировать заранее (то есть оно является вычисляемым). Учитывая априорную неопределенность развития вычислительного процесса на графе G_1 , данное D2 следует отнести к классу входных данных. Действительно, если вычисления “пойдут” по первому маршруту, то для него следует иметь начальное значение, если же вычисления будут развиваться по второму маршруту, то иницируемое значение будет заменено на вычисленное. В любом случае не возникнет ошибочная ситуация, связанная с отсутствием или неверным заданием начального значения для данного D2.

Таким образом, задача классификации данных для объектов-агрегатов связана с решением проблемы выделения всех независимых маршрутов, которые могут быть реализованы в графе агрегата.

2.3.2. Декомпозиция агрегатов. Алгебра трехзначной логики выделения классификационных признаков

Задача построения всех независимых маршрутов из корневой вершины в конечные вершины графа эквивалентна задаче декомпозиции исходного графа на совокупность частей графа, таких что $G = UR_j$, где R_j - ориентированные маршруты из корневой вершины в конечные.

Так например, граф, представленный на рисунке 2.11, может быть разложен на следующие маршруты (см. рис.2.12):

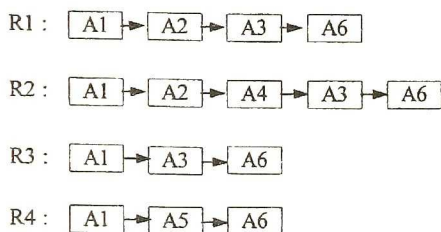


Рис. 2.12.

Каждый из представленных маршрутов является линейным графом (в общем случае они могут содержать и циклы), а развитие вычислительного процесса на графе происходит по одному из перечисленных направлений.

Для каждого из маршрутов можно построить паспорт на основе паспортов составляющих его объектов. Новый паспорт представляет собой теоретико-множественное объединение данных, входящих в маршрут объектов. Определенную трудность составляет задача классификации типов данных.

Абстрагируясь от содержательной части паспорта (наименований данных и их типов), рассмотрим задачу назначения классификационных признаков данным.

Классификационные признаки данных для линейных графов определяются расположением объектов на маршруте вычислений. Если несколько модулей на маршруте имеют общий параметр, то его классификационный признак будет определяться типом его первого вложения. Например, для маршрута $R1. A1 \rightarrow A2 \rightarrow A3 \rightarrow A6$ параметр D2 встречается в двух модулях: A2 и A3, в модуле A2 этот параметр является вычисляемым и не требует предварительной инициализации, а в модуле A3 — иницилируемым и должен принять значение заранее. Но, так как модуль A2 в этой цепочке встречается раньше, то D2 не нужно заранее инициализировать. Таким образом, на этом маршруте параметр D2 необходимо отнести к классу *вычисляемых*. С другой стороны, если данное встречается впервые как иницилируемое (внешнее), то его следует отнести к классу *внешних данных*.

Введем операцию конкретизации классификационных признаков для линейных маршрутов Δ , применение ее к объектам маршрута при построении паспорта маршрута представим следующим образом:

$$P(R_i) = P(A_{i1}) \Delta P(A_{i2}) \Delta \dots \Delta P(A_{in}),$$

где $P()$ - паспорт соответствующего объекта.

После того как будут получены паспорта всех маршрутов, на следующем этапе классификации параметров граф-агрегата эти маршруты необходимо будет объединить. При этом надо учитывать следующее обстоятельство: если несколько маршрутов имеют общий параметр, то его классификационный признак будет определяться с учетом приоритетности предварительной инициализации. Предполагается, что признак иницируемости является более приоритетным по отношению к признаку вычислимости. Например, если параметр D2 на маршруте R1 определился как иницируемый (входной), а на маршруте R2 как вычисляемый, то для нормальной работы граф-агрегата в целом, этот параметр должен быть инициализирован заранее.

Классификация параметров граф-агрегата $P(G)$ будет определяться полученными паспортами всех маршрутов:

$$P(G) = P(M_1) \nabla P(M_2) \nabla \dots \nabla P(M_m),$$

где ∇ операция над данными объектов, реализующая объединение паспортов маршрутов.

Определим семантику для введенных операций. Пусть заданы два объекта A и B с соответствующими паспортами:

$$P(A) = \{D_A^{in}, D_A^{out}\}, \quad P(B) = \{D_B^{in}, D_B^{out}\}.$$

Обозначим классификационные признаки данных следующим образом:

“0” - $d \notin P(A)$ - данное не принадлежит модулю A ;

“1” - $d \in D_A^{in}$ - данное принадлежит модулю A и является *входным* (иницируемым);

“2” - $d \in D_A^{out}$ - данное принадлежит модулю A и является *вычисляемым*.

Тогда таблица истинности операций *следования* Δ и *объединения* ∇ имеет вид (см. табл. 2.2):

Таблица 2.2

A	B	Δ	∇
0	0	0	0
0	1	1	1
0	2	2	2
1	0	1	1
1	1	1	1
1	2	1	1
2	0	2	2
2	1	2	1
2	2	2	2

Формально операции *следования* и *объединения* описываются следующим образом:

а) классификация данных в граф-агрегате на основании выполнения операции *следования*:

$$P(A \Delta B) = \{D_{A \Delta B}^{in}, D_{A \Delta B}^{out}\}, \text{ где}$$

$$D_{A \Delta B}^{in} = \left\{ d: d \in D_A^{in} \vee \left(d \notin P(A) \wedge d \in D_B^{in} \right) \right\}$$

$$D_{A \Delta B}^{out} = \left\{ d: d \in D_A^{out} \vee \left(d \notin P(A) \wedge d \in D_B^{out} \right) \right\}$$

б) классификация данных в граф-агрегате на основании объединения паспортов цепочек А и В (операция *объединения*):

$$P(A \nabla B) = \{D_{A \nabla B}^{in}, D_{A \nabla B}^{out}\}, \text{ где}$$

$$D_{A \nabla B}^{in} = \left\{ d: d \in D_A^{in} \vee d \in D_B^{in} \right\}$$

$$D_{A \nabla B}^{out} = \left\{ \begin{array}{l} d: \left(d \in D_A^{out} \wedge d \in D_B^{out} \right) \vee \\ \vee \left(d \notin P(A) \wedge d \in D_B^{out} \right) \vee \\ \vee \left(d \in D_A^{out} \wedge d \notin P(B) \right) \end{array} \right\}$$

Рассмотрим основные свойства заданных операций. Для наглядности, вместо символов, обозначающих паспорта объектов ($P(A)$, $P(B)$ и т.д.), будем применять символы, обозначающие сами объекты (A , B и т.д.).

1. Ассоциативность операций *следования* и *объединения*:

$$A \Delta (B \Delta C) = (A \Delta B) \Delta C = A \Delta B \Delta C = ABC$$

$$A \nabla (B \nabla C) = (A \nabla B) \nabla C = A \nabla B \nabla C$$

2. Коммутативность для операции *объединения*:

$$A \nabla B = B \nabla A, \text{ но } A \Delta B \neq B \Delta A$$

3. Операция *следования* дистрибутивна слева относительно операции *объединения*:

$$A \Delta (B \nabla C) = (A \Delta B) \nabla (A \Delta C)$$

4. Идемпотентность операций *следования* и *объединения*:

$$A \nabla A = A$$

$$A \Delta B \Delta A = A \Delta B$$

5. Поглощение для операции *следования* относительно операции *объединения*:

$$(A \Delta B \Delta C) \nabla (A \Delta B) = A \Delta B \Delta C$$

6. Отбрасывание общего окончания для операции *следования*:

$$(A \Delta B \Delta C) \nabla (A \Delta C) = (A \Delta B) \nabla (A \Delta C)$$

7. Свойство симметричного поглощения для операции *следования*:

$$(A \Delta C) \nabla (B \Delta C) \nabla (A \Delta B) = (A \Delta C) \nabla (B \Delta C) \nabla (B \Delta A) = (A \Delta C) \nabla (B \Delta C)$$

Представленные свойства операций *следования* и *объединения* позволяют оптимизировать алгоритм классификации данных за счет сокращения

количества маршрутов и уменьшения их длины. Рассмотрим возможный способ применения этих свойств на примере графа G_1 (рис. 2.11.). На первом этапе для каждого из четырех маршрутов графа (см. рис.2.12) строятся паспорта, при этом используется операция следования:

$$P(R_1) = A_1 \Delta A_2 \Delta A_3 \Delta A_6,$$

$$P(R_2) = A_1 \Delta A_2 \Delta A_4 \Delta A_3 \Delta A_6,$$

$$P(R_3) = A_1 \Delta A_3 \Delta A_6,$$

$$P(R_4) = A_1 \Delta A_5 \Delta A_6.$$

На втором этапе, полученные паспорта цепочек объединяются, образуя паспорт данных граф-агрегата:

$$\begin{aligned} P(G_1) &= P(R_1) \nabla P(R_2) \nabla P(R_3) \nabla P(R_4) = \\ &= (A_1 \Delta A_2 \Delta A_3 \Delta A_6) \nabla (A_1 \Delta A_2 \Delta A_4 \Delta A_3 \Delta A_6) \nabla \\ &\nabla (A_1 \Delta A_3 \Delta A_6) \nabla (A_1 \Delta A_5 \Delta A_6). \end{aligned} \quad (2.4)$$

Полученное выражение можно существенно сократить, если применить к нему преобразования, основанные на свойствах операций, в частности, свойство 6 отбрасывание общего окончания для операции следования. Очевидно, что выражения для цепочек R_1 и R_2 :

$$(A_1 \Delta A_2 \Delta A_3 \Delta A_6) \text{ и } (A_1 \Delta A_2 \Delta A_4 \Delta A_3 \Delta A_6)$$

имеют общее начало $(A_1 \Delta A_2)$ и общее окончание $(A_3 \Delta A_6)$. На основании свойств 1 и 6 можно преобразовать к виду:

$$\begin{aligned} (A_1 \Delta A_2 \Delta A_3 \Delta A_6) \nabla (A_1 \Delta A_2 \Delta A_4 \Delta A_3 \Delta A_6) \nabla (A_1 \Delta A_3 \Delta A_6) \nabla \\ \nabla (A_1 \Delta A_5 \Delta A_6) = (A_1 \Delta A_2 \Delta A_3 \Delta A_6) \nabla (A_1 \Delta A_2 \Delta A_4) \nabla (A_1 \Delta A_3 \Delta A_6) \nabla \\ \nabla (A_1 \Delta A_5 \Delta A_6). \end{aligned}$$

Таким образом, применяя последовательно свойства 1, 6 и 4 (поглощение следования относительно объединения), имеем:

$$P(G_1) = (A_1 \Delta A_2 \Delta A_4) \nabla (A_1 \Delta A_3 \Delta A_6) \nabla (A_1 \Delta A_5 \Delta A_6). \quad (2.5)$$

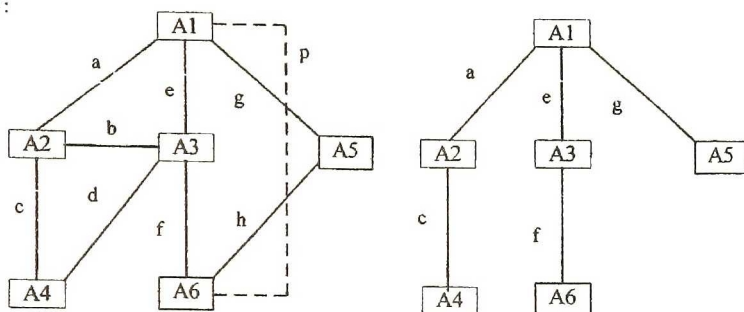
2.3.3. Построение множества вычислительных маршрутов

При построении множества ориентированных маршрутов на графе воспользуемся методом выделения базисных циклов [14]. Для этого из концевых вершин в корневую “заведем” временные дуги. Цель такого преобразования графа заключается в преобразование исходного объекта к виду, когда он может быть проанализирован методами цикломатики. Кроме этого, временно откажемся от ориентированности графа, заменив дуги ребрами. Причем парные дуги, имеющие разное направление, но одинаковые вершины, заменяются одним ребром. Все дуги пометим.

На рисунке 2.13а) показано преобразование графа G_1 к неориентированному виду $\tilde{G}_1$ предложенным выше способом.

Для графа $\tilde{G}_1$ построим его остов дерево D связный граф, не содержащий ни одного цикла (см. рис 2.13 б)).

$\tilde{G}_1$:



а) неориентированный граф

б) остов D графа $\tilde{G}_1$

Рис. 2.13.

Определим множество хорд остова D как множество ребер графа $\tilde{G}_1$, не принадлежащих остова D . В нашем случае это ребра $\{p, b, d, h\}$.

Введем цикломатическую матрицу $C(\tilde{G}_1)$, каждый элемент строки которой определяется следующим образом:

$$c_{ij} = \begin{cases} 1, & \text{если } j\text{-е ребро входит в } i\text{-й цикл;} \\ 0 & \text{- в противном случае.} \end{cases}$$

Множество $C(\tilde{G}_1)$ всех векторов, каждый из которых соответствует одному циклу графа, образует векторное пространство циклов графа $\tilde{G}_1$, линейное относительно операции поразрядного сложения по модулю два $\oplus$. В пространстве циклов графа имеется базис, состоящий из простых циклов. Причем любой цикл графа $\tilde{G}_1$ может быть представлен в виде линейной комбинации базисных векторов.

Из теоремы Эйлера следует, что число базисных циклов графа постоянно и равно его цикломатическому числу. Для графа $\tilde{G}_1$ цикломатическое число равно $\nu(\tilde{G}_1) = m - n + 1 = 9 - 6 + 1 = 4$. Из чего следует, что размерность пространства циклов графа $\tilde{G}_1$ равно четырем.

Построить базисную систему циклов можно методом, предложенным в [14].

Так базисную систему циклов для остового дерева D графа $\tilde{G}_1$ образует множество всех циклов графа $\tilde{G}_1$, каждый из которых содержит только одну хорду D .

Например, для рассматриваемого примера базис образуют циклы R_1, R_2, R_3, R_4 .

Выполняя $2^v - v - 1$ раз операцию сложения по модулю два над базисными циклами, получаем все множество циклов графа $\tilde{G}_1$.

$$C(\tilde{G}_1) = \left\| \begin{array}{ccccccccc} p & b & d & h & a & c & e & f & g \\ \hline 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 1 \end{array} \right\| \begin{array}{l} R_1 \\ R_2 \\ R_3 \\ R_4 \end{array}$$

Однако, с точки зрения построения маршрутов из корневой вершины в конечную, нас интересуют только те циклы, которые содержат фиктивное ребро p и ребра, исходящие из корневой вершины. Таких циклов восемь: R_1 , $R_5 = R_1 \oplus R_2$, $R_6 = R_1 \oplus R_3$, $R_7 = R_1 \oplus R_2 \oplus R_4$, $R_8 = R_1 \oplus R_3 \oplus R_4$, $R_9 = R_1 \oplus R_4$, $R_{10} = R_1 \oplus R_2 \oplus R_3$, $R_{11} = R_1 \oplus R_2 \oplus R_3 \oplus R_4$. Все циклы в графе $\tilde{G}_1$ можно подсчитать по формуле $2^v - v - 1 = 11$ ($v = 4$).

Докажем очевидное утверждение.

□ Утверждение 1. Множество несоставных циклов суграфа $G(V)$ неориентированного графа G содержится во множестве несоставных циклов самого графа G .

Действительно, цикломатическое число суграфа $G(V)$ $v' = m' - n + 1$, полученного из графа путем удаления некоторых ребер, меньше цикломатического числа графа G : $v = m - n + 1$, так как $m' < m$. Тогда по теореме Эйлера число базисных циклов суграфа меньше числа базисных циклов самого графа. Более того, множество базисных циклов суграфа содержится во множестве базисных циклов графа.

Действительно, пусть суграф имеет цикл, не принадлежащий графу. Следовательно, среди ребер цикла суграфа имеются ребра (хотя бы одно ребро), не принадлежащие графу. Последнее невозможно, поскольку множество ребер суграфа содержится во множестве ребер графа.

Так как базисные вектора суграфа содержатся в базисе графа, то множество циклов суграфа является подпространством циклов графа, из чего следует справедливость доказываемого утверждения. ■

Каждому неориентированному графу G поставим в соответствие ориентированный граф G' следующим образом: в графе G каждое ребро заменим парой дуг противоположной ориентации. Очевидно, что число циклов в графе удвоится по сравнению с числом циклов графа G . Точнее, граф G' будет содержать все циклы графа G , плюс те же циклы, но с противоположенным направлением обхода вершин графа. В неориентированном графе циклы перечисляются топологически, без учета направления обхода вершин цикла, хотя всегда существует два направления

обхода цикла. Если направление обхода принципиально, как в нашем случае при назначении классификационных признаков, то следует считать, что мы имеем двойной набор циклов.

Из сформулированного утверждения следует, что множество циклов суграфа ориентированного графа G' содержится во множестве циклов самого графа.

Последнее означает, что циклы преобразованного агрегата (когда из конечных вершин в корневую “заводятся” фиктивные дуги) содержатся во множестве циклов неориентированного аналога агрегата.

Для проверки принадлежности цикла ориентированному агрегату построим схемы маршрутов (циклов без фиктивных дуг) следующим образом: дуга включается в маршрут с положительным знаком, если она исходит из текущей вершины маршрута, и с отрицательным знаком, если она входит в вершину маршрута.

Считается, что маршрут принадлежит агрегату, если все дуги маршрута имеют положительный знак.

Таблица 2.3

№	Цикл	Маршрут
1	R1	$M1=\{e, f\}$
2	R10	$M2=\{e, -b, c, d, f, \}$
3	R5	$M3=\{a, b, f\}$
4	R6	$M4=\{a, c, d, f\}$
5	R9	$M5=\{g, h\}$

В таблице 2.3 рассмотрены только те циклы из восьми найденных для графа $G1$ (R1, R5, R6, R7, R8, R9, R10, R11), которые содержат только одну дугу, исходящую из корневой вершины, поскольку все маршруты агрегата должны исходить из корневой вершины только один раз. Из таблицы видно, что только один из приведенных маршрутов ($M2$) не удовлетворяет условию положительности дуг маршрута. Отбросив маршрут $M2$, имеем четыре маршрута агрегата $G1$.

К сожалению, количество потенциальных маршрутов графа комбинаторно растет с ростом цикломатического числа ν , характеризующего сложность программного модуля [21]. Из таблицы 2.4 видно, что, начиная с $\nu=6$, число потенциальных маршрутов полносвязного графа исчисляется десятками, а далее - сотнями тысяч. В этих условиях сравнительно простая процедура проверки принадлежности маршрута ориентированному графу становится невыполнимой, не говоря уже о классификации типов данных агрегата, когда число данных в каждом из модулей агрегата может исчисляться сотнями или тысячами.

Для сокращения числа операций над данными в процессе их классификации предлагается использовать метод, основанный на аксиомах трехзначной логики.

Таблица 2.4

порядок графа	max число дуг	циклomat. Число	прогнозируемое число циклов
2	1	0	0
3	3	1	1
4	6	3	7
5	10	6	63
6	15	10	1023
7	21	15	32767
8	28	21	2097151
9	36	28	2.68E+08
10	45	36	6.87E+10

2.3.4. Сжатие числа операций процедуры классификации данных

Для реализации алгоритма сжатия количества операций предлагается использовать аксиомы 4, 5 и 6, которые сформулируем как правила вывода в форме метаформул:

$$\text{П1: } \frac{\alpha \Delta \beta \Delta \alpha}{\alpha \Delta \beta},$$

$$\text{П2: } \frac{(\alpha \Delta \beta \Delta \gamma) \nabla (\alpha \Delta \beta)}{\alpha \Delta \beta \Delta \gamma} \quad \text{правило поглощения,}$$

$$\text{П3: } \frac{(\alpha \Delta \beta \Delta \gamma) \nabla (\alpha \Delta \gamma)}{(\alpha \Delta \beta) \nabla (\alpha \Delta \gamma)} \quad \text{поглощение общего окончания.}$$

Последовательность вершин, лежащих на маршруте и связанных операцией следования, назовем условно "дизъюнктами". Для простоты записи и восприятия вместо оператора следования Δ в записи формулы будем употреблять "запятую".

Алгоритм упрощения формулы классификации данных состоит из следующих шагов:

1. Все дизъюнкты упорядочиваются, в первую очередь, по длине и лексикографически, в случае одинаковой длины. Более длинные дизъюнкты ставятся на первое место.

2. К каждому дизъюнкту применяется правило П1 до тех пор, пока это возможно.

3. К каждому дизъюнкту, начиная с первого, применяется правило П3. Причем, контактный "партнер" ищется на множестве дизъюнктов меньшей

длины. Если правило ПЗ реализуется, то множество дизъюнктов переупорядочивается.

4. После выполнения пункта 3 алгоритма для каждого дизъюнкта меньшей длины ищется дизъюнкт, его поглощающий, т.е. применяется правило П2. Поглощаемый дизъюнкт вычеркивается из списка дизъюнктов.

Работу предложенного алгоритма рассмотрим на примере упрощения формулы классификации данных агрегата $G1$:

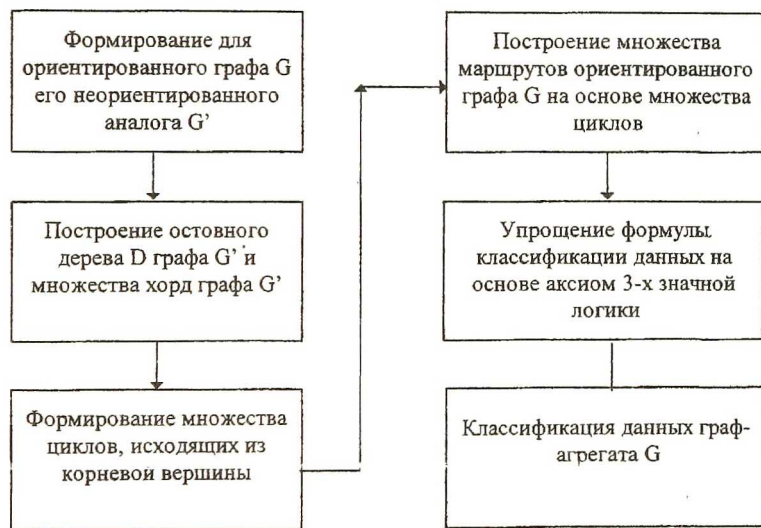
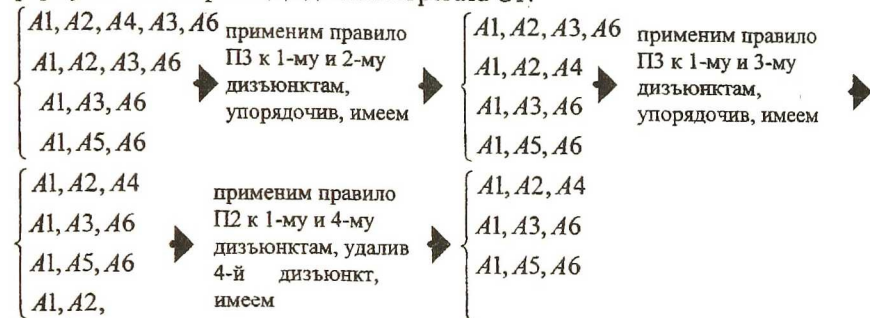


Рис.2.14 Функциональная схема классификации данных агрегата

В результате исходная формула (2.4) упростилась до вида (2.5), количество операций сократилось с 14 до 8. Вычислительные эксперименты показали, что сокращения числа операций на реальных объектах в среднем составляет 50%, причем, с ростом цикломатического числа, когда число

маршрутов нарастает, производительность алгоритма сжатия операций увеличивается.

Общая функциональная схема алгоритма классификации данных агрегата представлена на рисунке 2.14.

2.3.5. Алгоритм классификации данных на основе частичного перебора маршрутов ориентированного графа

Описанный выше алгоритм классификации данных агрегата использует цикломатический анализ неориентированных графов, эквивалентных исходному. Главной проблемой этого метода является комбинаторный рост числа циклов с ростом цикломатического числа графа (см. табл.2.4) и ограниченность типов рассматриваемых циклов. Цикломатика обеспечивает перечисление простых и сложных циклов, не рассматривая составные циклы, когда одна или несколько дуг повторяются. На рисунке 2.15 показан граф, имеющий составной цикл $R=\{a, b, c, a, d\}$.

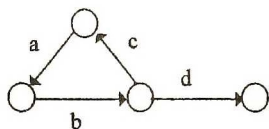


Рис. 2.15

В работах [26, 34] предлагается метод классификации типов данных, основанный на реализации частичного перебора маршрутов граф-агрегата.

На этапе перечисления маршрутов из корневой вершины в

концевые, метод использует свойства алгебры 3-х значной логики классификации данных.

Маршруты ориентированного графа будем описывать списками вершин. Данный подход менее удобен, чем при использовании списка дуг, поскольку часто содержит повторяющиеся вершины. Например, маршрут $M=\{A, B, E, F, B, C, D, B, G\}$ графа G_2 (см. рис. 2.16) повторяет вершину B три раза.

G_2 :

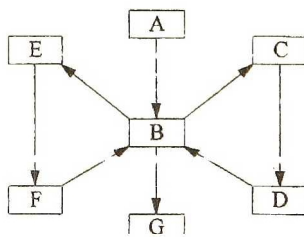


Рис. 2.16.

Однако учитывая свойство 4 операции следования алгебры 3-значной логики, все “повторные” вхождения вершин графа в список можно исключить.

Введем понятие *схемы маршрута*, как списка вершин маршрута без повторений.

С точки зрения классификации данных агрегата *схемы маршрутов* определяют дизъюнкты алгебры 3-х значной логики. С другой стороны, введенное понятие удобно для организации алгоритма перечисления маршрутов орграфов, поскольку именно на схемах маршрутов появляется возможность реализации возврата к ранее рассмотренным вариантам и осуществления поиска новых вариантов развития вычислительного процесса на орграфе.

Свободными вершинами графа G относительно схемы маршрута S будем называть вершины графа, не содержащиеся в схеме маршрута S , т.е. $L(S) = V(G) \setminus V(S)$, где $V(G)$ и $V(S)$ соответственно вершины графа и схемы маршрута.

Алгоритм частичного перебора (АЧП) использует следующие правила:

1. Вершины графа кодируются символами упорядоченного множества любым способом, но так, чтобы корневая вершина имела наименьший код, а концевые - наибольшие значения.

2. На каждом шаге работы алгоритма для текущего состояния схемы маршрута S_i ищется путь на орграфе перехода из последней вершины S_i в любую свободную вершину схемы S_i .

3. Если переход *возможен*, то схема маршрута “расширяется” на один символ.

4. Если переход *невозможен*, то рассматривается следующая вершина списка свободных вершин и так до тех пор, пока не кончится список вершин.

5. Если переход из текущей вершины в любую из свободных вершин (не содержащихся в схеме маршрута) *невозможен*, то происходит “откат” по схеме маршрута на один символ назад.

6. При достижении концевой вершины алгоритм “откатывает” на один символ назад.

7. Алгоритм завершает свою работу, если список свободных вершин корневой вершины исчерпан.

Схема маршрута $L = \{v_0, v_1, \dots, v_m\}$ реализует сжимающее кодирование маршрутов на орграфах, поскольку описывает путь из корневой вершины в концевую $v_0 \rightarrow v_m$ через любое количество повторений, предшествующих v_m вершин. Схема маршрута позволяет закодировать любые маршруты орграфа: простые, сложные и составные.

Рассмотрим наиболее важные свойства схем маршрутов.

□ **Лемма 1.** $(\forall \alpha \in N \quad (A, B_{i_1}, A, B_{i_2}, \dots, B_{i_\alpha}, A, \dots)) \rightarrow (A, B_{i_1}, B_{i_2}, \dots, B_{i_\alpha}, \dots)$

т.е. любое (бесконечное) число вхождений вершины A в составной маршрут может быть сокращено до одного (первого) вхождения на схеме маршрута.

Пусть $\alpha = 1$, тогда из свойства 4 алгебры 3-х значной логики следует $A, B, A \rightarrow A, B$. Для $\alpha = 2$, применив свойство 4 два раза имеем: $A, B_1, A, B_2, A \rightarrow A, B_1, A, B_2 \rightarrow A, B_1, B_2$. Очевидно, что если свойство справедливо для всех $n < \alpha$, то оно справедливо и для α , т.е. $(\forall n < \alpha \quad (A, B_{i_1}, A, B_{i_2}, \dots, B_{i_n}, A)) \rightarrow (A, B_{i_1}, B_{i_2}, \dots, B_{i_\alpha})$. Но тогда в

соответствии с аксиомой трансфинитной индукции, это свойство будет справедливо для любого α . ■

Лемма 2. $(\forall \alpha \in N \quad (A, B_{i_1}, B_{i_2}, \dots, B_{i_\alpha}, A, \dots)) \rightarrow (A, B_{i_1}, B_{i_2}, \dots, B_{i_\alpha}, \dots)$ т.е.

повторное вхождение вершины A в составной маршрут может быть сокращено до одного (первого) вхождения на схеме маршрута.

Пусть $\alpha = 1$, тогда из свойства 4 алгебры 3-х значной логики следует $A, B, A \rightarrow A, B$. Для $\alpha = 2$, применив свойство 4 два раза имеем: $A, B_1, (B_2, A) \rightarrow A, B_1, (A, B_2, A) \rightarrow (A, B_1, A), B_2 \rightarrow A, B_1, B_2$. Очевидно, что если свойство справедливо для всех $n < \alpha$, то оно справедливо и для α , т.е. $(\forall n < \alpha \quad (A, B_{i_1}, B_{i_2}, \dots, B_{i_n}, A)) \rightarrow (A, B_{i_1}, B_{i_2}, \dots, B_{i_\alpha})$. Но тогда в

соответствии с аксиомой трансфинитной индукции это свойство будет справедливо для любого α . ■

Лемма 3. Мощность множества составных маршрутов на конечном полносвязанном орграфе несчетно и равна 2^{N_0} .

Составной маршрут орграфа, имеющего бесконечный цикл, содержит бесконечное число повторений вершины или комбинаций вершин. Пусть граф состоит из двух вершин, которые закодируем цифрами 0 и 1. Полносвязанный граф содержит любой составной маршрут, составленный из любых сочетаний 0 и 1, например, 1001110... . Если перед маршрутом поставить десятичную точку, то получим код десятичного числа в двоичном представлении $x=0.1001110... .$ Очевидно, что таким способом можно закодировать все действительные числа, принадлежащие интервалу $(0,1)$, а таких чисел, как известно, несчетно, следовательно, несчетно и число составных маршрутов. Увеличение степени полносвязанного графа изменяет лишь основу представления чисел, следовательно оно справедливо для любых полносвязанных орграфов. ■

Утверждение 2. Множество всех схем составных маршрутов полносвязанного орграфа конечно.

Леммы 1 и 2, утверждают, что из любого бесконечного составного маршрута можно удалить все повторные вхождения вершин, вплоть до первого вхождения. Но тогда схема маршрута содержит только не повторяющиеся вершины. Для конечного графа порядка n длина схемы маршрута не превышает

п. Более того, общее число схем маршрутов на конечном алфавите не может быть более чем $n!$ (числа перестановок символов вершин).

Таким образом, введения понятия схем маршрутов позволяет осуществить отображение из потенциально бесконечного множества маршрутов на конечное множество их схем.

Покажем, что алгоритм АЧП реализует частичный перебор схем маршрутов. В качестве примера рассмотрим граф G_1 , вершины которого пронумеруем так, как это показано на рисунке 2.17. Алгоритм АЧП формирует множество схем маршрутов, реализуя разбиение исходной задачи на взаимно независимые подзадачи в соответствии со стратегией И/ИЛИ-графов [4]. Построим мультиграф с двумя типами вершин. Вершины типа "И" помечаются номерами вершин исходного графа, вершины типа "ИЛИ" помечаются списками вершин, достижимых (смежных) и недостижимых из

G_1 :

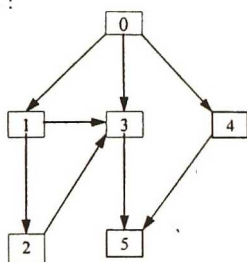


Рис. 2.17.

соответствующей "И"-вершины графа. Правила алгоритма АЧП позволяют на каждом шаге алгоритма множество свободных вершин разбить на две части: достижимое и недостижимое подмножества. Причем достижимость понимается либо как непосредственный переход по дуге графа, либо опосредованно, через вершины предшествующего уровня иерархии, т.е. в соответствии со схемой маршрута.

На рисунке 2.18 а) показан И/ИЛИ-граф алгоритма АЧП для графа G_1 . Решением задачи перечисления схем маршрутов является подграф И/ИЛИ-графа, из которого исключены ИЛИ вершины (см. рис. 2.18 б)).

2.3.6 Эффективность алгоритма АЧП

Эффективность зависит от связности графа. Для полносвязанного графа реализуется полный просмотр всех схем маршрутов. В остальных случаях число проверок реализуемости маршрутов на графе ξ_G равно сумме мощностей множеств достижимости и недостижимости. Для случая, рассмотренного в примере, таких проверок было произведено 25 и обнаружено 4 схемы, в то время как общее число проверок $5! = 120$. ξ_G для некоторого графа G , можно оценить по формуле :

$$\xi_G = \sum_{k=1}^n \xi_k,$$

где ξ_k — мощность множества достижимости или недостижимости И/ИЛИ-графа.

эта метрика со сложностью алгоритма построения неперiodических маршрутов, определяемой с помощью топологического дерева. С этой целью проведем эксперимент, который заключается в построении полного множества графов заданного порядка и оценки их сложности с помощью метрики Маккейба.

Для проведения эксперимента реализован алгоритм, который порождает графы, удовлетворяющие определенным условиям. Эти условия накладываются концепциями технологиями графо-символического программирования и подразумевают наличие в графе корневой и конечной вершин. Алгоритм порождения графов основан на методе частичного перебора из полного комбинаторного множества всех существующих графов заданного порядка.

Полученные результаты показывают, что в среднестатистическом смысле сложность алгоритма построения неперiodических маршрутов зависит от топологической сложности графа. Однако, при рассмотрении конкретных случаев имеется большой разброс в показателе сложности алгоритма от топологической сложности.

Зависимость разброса усредненной сложности алгоритма АПЧ в зависимости от степени графа получена экспериментально, в результате моделирования орграфов различной структуры (см. рис.2.19).

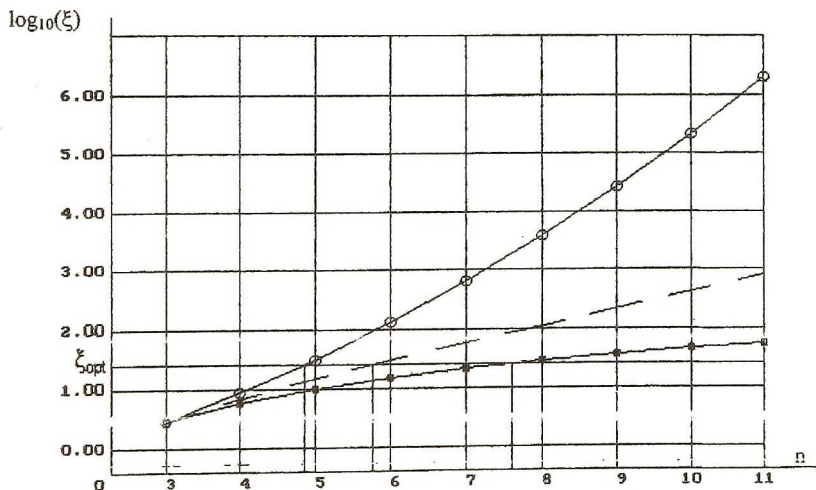


Рис 2.19.

Просматривались варианты от линейных графов (нижняя линия) до полностью связанных графов (верхняя линия). “Средняя” линия на рисунке разбивает приблизительно пополам множество топологически разных графов. Около этой линии “концентрируется” наибольшее число топологически разных графов одинакового порядка. Из рисунка видно, что с ростом n сложность алгоритма АЧП увеличивается даже для линейных графов, и рациональным порядком графа можно считать $n \in [6,8]$. Примерно этого же уровня сложности следует придерживаться при разработке граф-агрегатов.

2.4. Структурная оптимизация агрегатов

Основной проблемой классификации данных агрегатов и их тестирования является комбинаторный рост маршрутов агрегата с ростом порядка графа. Число несоставных маршрутов орграфа косвенно зависит от цикломатического числа графа, характеризующего структурную сложность программы. В общем случае существует корреляция между цикломатической сложностью графа и интуитивным ощущением сложности программы. Программа с цикломатической сложностью более 10 создает впечатление ненадежной [18]. С другой стороны, хорошо структурированная программа более ясно выражена, ее легче читать, она более понятна, более легка для проверки и отладки.

Рассмотрим агрегаты с одной корневой вершиной и одной тупиковой. Если ввести дополнительную (фиктивную) дугу из тупиковой вершины в корневую, то, как было показано в п. 2.3.3, структурную сложность агрегата можно анализировать методами цикломатики неориентированных графов. Причем число независимых циклов совпадает с цикломатическим числом графа $\nu = m - n + 2$, где m - число дуг графа, n - порядок графа. Общее число циклов можно подсчитать по формуле $N_{\Sigma} = 2^{\nu} - 1$. В то же время, нас интересуют только те циклы, которые содержат фиктивную дугу, поскольку ее удаление порождает все маршруты из корневой вершины в тупиковую. Таких маршрутов будет $N_M = 2^{\nu-1} - 1$.

Можно показать, что число маршрутов тем меньше, чем больше дуг, исходящих из корневой вершины.

Рассмотрим пример из п. 2.3.3. Из цикломатической матрицы графа $\tilde{G}_1$ $C(\tilde{G}_1)$ выделим столбцы, принадлежащие дугам, исходящим из корневой вершины: $a = (0,1,1,0)^T$, $e = (1,1,1,1)^T$, $g = (0,0,0,1)^T$. Введем переменные x_1, x_2, x_3, x_4 , принимающие значения либо 0, либо 1. Составим систему уравнений:

$$\begin{cases} c_{1i_1}x_1 \oplus c_{2i_1}x_2 \oplus c_{3i_1}x_3 \oplus c_{4i_1}x_4 = b_1 \\ c_{1i_2}x_1 \oplus c_{2i_2}x_2 \oplus c_{3i_2}x_3 \oplus c_{4i_2}x_4 = b_2 \\ c_{1i_3}x_1 \oplus c_{2i_3}x_2 \oplus c_{3i_3}x_3 \oplus c_{4i_3}x_4 = b_3, \end{cases} \quad (2.6)$$

где b_1, b_2, b_3 - свободные члены, c_{ij} - коэффициенты цикломатической матрицы при дугах, исходящих из корневой вершины. Можно показать, что над уравнениями системы (2.6) можно производить эквивалентные преобразования перестановки любых двух уравнений и сложение по модулю любых двух уравнений. Этих преобразований достаточно, чтобы в арифметике

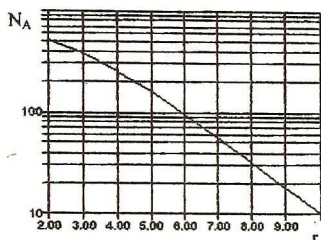


Рис.2.20

“сложения по модулю два” матрицу системы уравнений усеченным методом Гаусса привести к диагональному виду. Очевидно, что ранг системы (2.6) не может быть больше, чем число уравнений или число дуг r , исходящих из корневой вершины.

В технологии ГСП развитие вычислительного процесса в каждый конкретный момент времени может идти

только по одному из направлений, исходящих из корневой вершины. Если ввести столбец свободных членов вида $(1, 0, 0)$, то с помощью системы уравнений будут найдены все несоставные маршруты, проходящие через первую дугу, исходящую из корневой вершины (а).

Число таких циклов можно оценить величиной 2^{v-r} , где $v-r$ число свободных переменных системы (2.6). Учитывая количество систем уравнений, описывающих прохождение маршрутов через все дуги исходящие из корневой вершины, имеем оценку числа несоставных маршрутов агрегата $N_A = r2^{v-r}$.

На рисунке 2.20 показана зависимость числа маршрутов N_A от количества исходящих из корневой вершины дуг для графов с цикломатическим числом $v=10$. Из рисунка видно, что чем больше дуг исходит из корневой вершины, тем меньше получается несоставных маршрутов. В связи с чем, при разработке агрегатов, можно рекомендовать наибольшее количество проверок переносить в корневую вершину.

При разработке агрегатов ПОП возможны ситуации, когда может быть превышено разумное для отладки и классификации типов данных значение цикломатического числа агрегата, что приведет к значительным затратам ресурсов ЭВМ при тестировании модуля. Проблема может быть решена за счет структурной оптимизации граф-агрегата путем иерархического его разбиения на несколько агрегатов с меньшей структурной сложностью. Рассмотрим метод структурной оптимизации и условия ее реализации.

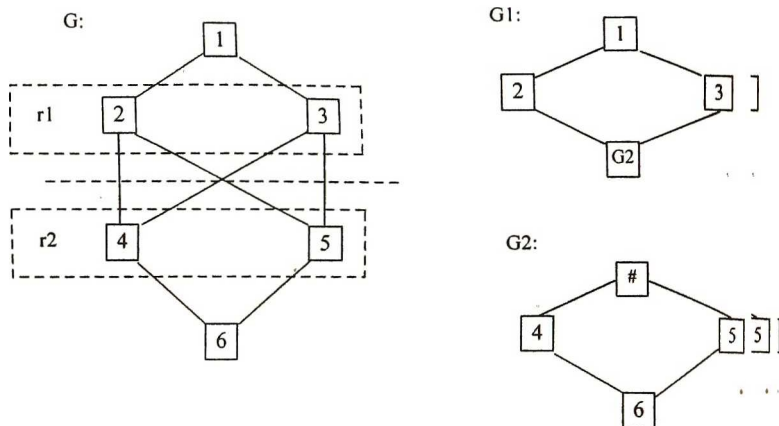


Рис.2.21

На рисунке 2.21 схематично представлен агрегат G, который разделим по линии разреза на два агрегата G1 и G2 следующим образом:

1. В агрегаты входят все вершины исходного агрегата до линии разреза плюс дополнительные вершины - "связки" (в примере вершины, помеченные "#" и "G2"). Причем множество вершин исходного графа распадается на два непересекающихся подмножества V_1 и V_2 .

2. Каждый из полученных агрегатов содержит все дуги, соединяющие его вершины из исходного агрегата за вычетом дуг разреза.

3. Дополнительные вершины связываются с теми вершинами подграфа (подмножества V_1 или V_2), в которые входили, или из которых исходили дуги разреза. Причем в подграфе, содержащем корневую вершину исходного графа, дополнительная вершина является тупиковой, если отсутствуют дуги из множества V_2 в V_1 и еще одна дополнительная концевая вершина, если такие дуги есть. В подграф, не содержащий корневую вершину, добавляется одна или две дополнительные вершины, одна из которых является корнем.

4. В паспорта каждого из новых агрегатов вносятся дополнительные данные (S_{G1} и S_{G2}), фиксирующие текущие состояния агрегатов (имя или номер вершины, смежной с "дополнительной" вершиной подграфов, перед переходом в нее).

5. Предикаты на новых дугах, связанные с дополнительными вершинами, составляются следующим образом:

- дуги, исходящие из вершины # подграфа G2, помечаются предикатами вида $(P_{ij} \& (S_{G1} = i)) \vee (P_{kj} \& (S_{G1} = k)) \vee \dots$, где P_{ij} - предикат, помечающий дугу, исходящую из вершины i в вершину j ; $i, k, \dots$ - вершины подграфа G1, "смежные" вершине j подграфа G2 (связанные дугами разреза);

- аналогично поступают с новыми дугами подграфа G1, если из подграфа G2 имеются обратные связи.

новыми дугами подграфа G1, если

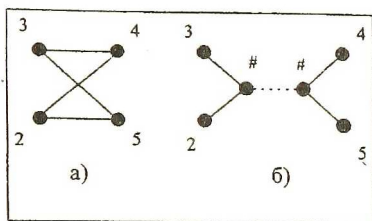


Рис.2.22.

Несмотря на то, что топология новых графов сильно изменилась (см. рис.2.21), можно показать, что развитие вычислительного процесса на композиции графов не меняется. Действительно, изменения в структуре графа вносились по линии “разреза”, причем двудольный граф разреза (см. рис. 2.22 а)) трансформировался в “звезду” (см. рис. 2.22 б)).

Реальные переходы исходного графа на разрезе описываются множеством пар $\{(3,4),(3,5),(2,4),(2,5)\}$, которые на графе G2 описываются множеством условных переходов $\{(\#,4|3),(\#,5|3),(\#,4|2),(\#,5|2)\}$, связанных через вершину # с вершинами 2, 3 графа G1. Например, маршрут 3,5 на композиции графа представляется маршрутом 3, #, #, (5|3). Поскольку вершины # чисто технологические и никак не влияют на данные ПОП, то цепочка 3, #, #, (5|3) свертывается до 3,5. На агрегате G2 условный переход из вершины # в вершину 4 кодируется предикатом $(P_{34} \& (S_{G1} = 3)) \vee (P_{24} \& (S_{G1} = 2))$, а условный переход из вершины # в вершину 5 кодируется предикатом $(P_{35} \& (S_{G1} = 3)) \vee (P_{25} \& (S_{G1} = 2))$.

Таким образом, структурные преобразования графа G “сворачиваются” за счет более сложных логических условий предикатов. Платой за структурные упрощения графа является усложнение предикативной части агрегата.

Определим условия, при которых целесообразно производить декомпозиционное разбиение граф-агрегата.

Суть метода рассмотрим на примере, представленном на рисунке 2.21. Пусть $\nu_{\Sigma} = m_{\Sigma} - n_{\Sigma} + 1$ - цикломатическое число исходного графа G, где m_{Σ}, n_{Σ} - соответственно число дуг и вершин графа. Пусть n_1 и n_2 - количество вершин в графах G1 и G2. Причем справедливо

$$n_{\Sigma} + 2 = n_1 + n_2 \quad (2.7)$$

Пусть $\tilde{\nu} = \tilde{m} - \tilde{n} + k$ - цикломатическое число разреза (в общем случае многосвязанного двудольного графа). $\tilde{r}_1$ и $\tilde{r}_2$ - число вершин в подмножествах разреза V_1 и V_2 ($\tilde{n} = \tilde{r}_1 + \tilde{r}_2$). Дуги графов связаны балансовым соотношением:

$$m_{\Sigma} = m_1 + m_2 + \tilde{m} \quad (2.8)$$

Подсчитаем цикломатические числа графов G1 и G2:

$$\nu_1 = (m_1 + \tilde{r}_1) - n_1 + 1 \text{ и } \nu_2 = (m_2 + \tilde{r}_2) - n_2 + 1.$$

Тогда

$$\nu_1 + \nu_2 = m_1 + m_2 + \tilde{r}_1 + \tilde{r}_2 - (n_1 + n_2) + 2 = \left| \begin{array}{l} \text{учитывая балансовые} \\ \text{уравнения (2.7)-(2.8)} \end{array} \right| =$$

$$\begin{aligned}
 &= (m_{\Sigma} - \tilde{m}) + \tilde{n} - n_{\Sigma} = (m_{\Sigma} - n_{\Sigma} + 1) - (\tilde{m} - \tilde{n} + k) = \\
 &= v_{\Sigma} - \tilde{v}.
 \end{aligned}$$

Таким образом, сумма цикломатических чисел подграфов равна цикломатическому числу исходного графа за вычетом цикломатического числа разреза:

$$v_1 + v_2 = v_{\Sigma} - \tilde{v}. \quad (2.9)$$

В нашем случае $v_1 + v_2 = 8 - 6 + 1 - (4 - 4 + 1) = 2$, в то время как исходный граф имеет цикломатическое число равное 3, декомпозиция графа дает положительный результат. Формула (2.9) по существу является критерием оценки эффективности разреза, и с ее помощью можно подбирать наиболее рациональные варианты разреза графа. Например, для графа G еще для трех топологически разных разрезов, (отделяющих вершины 2-4, 1-2 и 1), цикломатические числа разрезов оцениваются величинами: $v_1 = 4 - 6 + 2 = 0$; $v_2 = 3 - 5 + 2 = 0$; $v_3 = 2 - 3 + 1 = 0$, следовательно, представленные разрезы не уменьшают структурную сложность графа.

Полученную формулу можно обобщить на случай k разрезов графа:

$$\sum_{i=1}^{k+1} v_i = v_{\Sigma} - \sum_{j=1}^k \tilde{v}_j. \quad (2.10)$$

Анализ формулы (2.10) показывает, что декомпозиция, понижающая сложность агрегата, реализуется в случае, если число дуг разреза плюс его связность больше числа вершин разреза $\tilde{m} + k > \tilde{n}$. Из чего следует, что любое количество разрезов линейного графа не уменьшает его структурной сложности.

Глава 3

АВТОМАТИЗАЦИЯ ТЕСТИРОВАНИЯ ОБЪЕКТОВ ТЕХНОЛОГИИ ГСП

Разработка любого программного продукта может быть представлена, как процесс, состоящий из ряда последовательных преобразований одного описания решаемой задачи в другое, начиная от постановки задачи и заканчивая программой, реализованной в кодах конкретной ЭВМ. Все время существования программного продукта от зарождения идеи по его созданию, до завершения его эксплуатации, обычно определяют как жизненный цикл. Укрупненно можно выделить пять наиболее важных этапов жизненного цикла программного продукта (ЖЦПП): спецификацию (10%), проектирование (10%), кодирование (10%), отладку (20%) и сопровождение (50%). В скобках записаны относительные затраты ресурсов на создание ПП, по данным работы [12].

По затратам времени, человеческих и машинных ресурсов все эти этапы не одинаковы. Наиболее “дорогими”, в этом смысле, являются этапы, связанные с поиском ошибок в программах. Затраты ресурсов на них могут быть равными, или даже превосходить совокупные затраты ресурсов на остальных этапах. Эти затраты быстро увеличиваются при возрастании требований к качеству ПП. По оценкам, приведенным в работе [38], на долю устранения ошибок и сопровождения ПП приходится почти 75% всех затрат на разработку. Следует учитывать, что значительная часть работ, выполняемых на этапе сопровождения, связана с поиском и устранением оставшихся в программе ошибок.

Таким образом, приведенные цифры, позволяют сделать вывод, что наиболее дорогостоящими этапами ЖЦПП, с точки зрения затрат ресурсов, являются этапы отладки ПП.

Проблемам отладки и тестирования ПП посвящены многочисленные исследования отечественных и зарубежных специалистов. В конце 70-х начале 80-х годов состояние в этой области отражают публикации таких зарубежных исследователей как Майерс Г. [37,38], Гласс Р. [12], Маккейб Т., Бозм Б.[3] и других. Среди отечественных исследователей различными аспектами этой проблемы занимались Архангельский Б.В. [1], Ершов А.П., Штрик А.А., Ибгуду К.А., Бетелин В.Б. и др. Значительный вклад в исследование основных принципов и разработку промышленных методов автоматизации отладки ПП внес Липаев В В [35].

Основной целью тестирования является обнаружение и устранения программных ошибок или (и), если это возможно, доказательство отсутствия последних в программе.

Перечень современных методов и подходов к решению проблемы тестирования программ широк и многообразен. Это многообразие определяется, с одной стороны, существующей практикой использования ЭВМ

при решении разнообразных задач и, соответственно, специфическими особенностями самих программных продуктов. С другой стороны - более высоким порядком сложности задачи тестирования ПП по сравнению с задачей их разработки.

На начальных этапах разработки программ весьма эффективны *методы ручного тестирования* [37]. Эти методы позволяют обнаружить наиболее простые и массовые ошибки в программе. Однако основная часть методов тестирования в той или иной степени являются автоматизированными.

Можно выделить большую группу методов, которую часто классифицируют как *статические* методы [35]. Статические методы используют исходные тексты программ или информационные структуры, содержащие символьное описание тестируемых объектов. При статическом тестировании производится формализованный контроль корректности структуры программ, оценка длительностей исполнения программ, а также другие виды анализа корректности исходного текста. К этой группе методов непосредственно примыкают методы доказательства правильности программ [43].

Другую группу методов определяют как *динамическое тестирование*. При динамическом тестировании основные результаты получаются в процессе исполнения программы в машинном коде [35].

Среди динамических методов тестирования можно выделить две группы методов: *детерминированное* и *стохастическое* тестирование.

При *детерминированном* тестировании на вход объекта подаются конкретные значения тестовых параметров. Этим параметрам на выходе соответствуют конкретные значения эталонных результатов. Такой подход позволяет не только обнаружить ошибку в программе, но и, в ряде случаев, ее локализовать.

Стохастические методы тестирования предполагают использование исходных данных в виде совокупности случайных величин, описываемых своими распределениями. Этим распределениям исходных данных соответствуют эталонные распределения выходных значений. При этом оценка качества работы программы и результатов тестирования производится в зависимости от степени соответствия параметров полученных распределений эталонным значениям.

В главе рассматриваются методы тестирования и оценки надежности объектов технологии ГСП, учитывающие особенности разработки ПП в рамках ГСП.

3.1 Стохастическое тестирование базовых модулей

Вопрос целесообразности тестирования программ не столь очевиден, как это кажется на первый взгляд. Например, Дейкстра [56] приводит убедительные доводы в пользу бессмысленности тестирования программ, поскольку число возможных вариантов данных для алгоритма вообще

бесконечно, то никакая серия тестов не сможет никогда гарантировать правильность алгоритма. "...Тесты могут служить для демонстрации наличия ошибок в программе, но не их отсутствия...", и из этого делает вывод о "бесполезности" тестов и необходимости доказательств правильности программ.

С другой стороны, доводы Б. Мейера [43] в пользу тестирования не менее убедительны. Действительно, ни один программист не "пустит" программу в эксплуатацию, не подвергнув ее некоторому числу "холостых" выполнений, используя данные, для которых известен результат для проверки ее работоспособности. Кроме того, метод доказательства правильности программ тоже не является непогрешимым, поскольку *строгое доказательство имеет не меньшую вероятность содержать ошибки, чем сама программа.*

Не умаляя достоинств метода доказательства правильности программ, тем более, что в некоторых предметных областях (например, в области искусственного интеллекта и языков логического программирования) этот метод является единственно возможным, сосредоточим свое внимание на методах тестирования программ.

На наш взгляд полемика в пользу или против тестирования программ обусловлена недостаточностью накопленных знаний и методов в этой области, поскольку такая сравнительно молодая область деятельности человека, как программирование, до сих пор развивалась главным образом в направлении совершенствования техники и средств разработки программ, а к систематическому изучению "последствий" программирования приступили сравнительно недавно.

Техника тестирования программных продуктов во многом определяется их специализацией. Рассмотрим ПП, построенные на основе численного анализа и вычислительной математики. При построении таких программ в качестве "неделимых" программных единиц обычно выступают модули, реализующие некие функциональные преобразования из n -мерного пространства действительных или комплексных чисел. на m -мерное подпространство действительных (комплексных) чисел. В данном случае важным является то, что наблюдаемые переменные (зависимые переменные) функционально (количественно) зависят от "объясняющих" переменных (аргументов). К предметным областям с такой схемой построения исходных модулей относятся САПР технических объектов, АСНИ и т.д.

Очевидно, что существуют предметные области, в которых программные единицы устроены иначе. Например, модули, реализующие прорисовку на экране любых графических объектов, не содержат функциональных связей, и техника отладки таких модулей имеет свои специфические особенности.

В технологии ГСП в качестве элементарных программных единиц выступают *базовые модули*, на основе которых, путем привязки к данным

ПОП, строятся *акторы*. Акторы реализуют отображение $A_k: D_k^{in} \rightarrow D_k^{out}$ из множества входных параметров модуля на множество вычисляемых данных. В паспорте каждого актора содержится полное описание не только типов параметров, но и областей определения и значений актора. Область определения актора может быть описана либо с помощью алгоритма (например, с помощью системы неравенств), либо как декартово произведение доменов каждого из параметров

$$\Omega_{A_k} = \Omega_{d_1} \times \Omega_{d_2} \times \dots \times \Omega_{d_{n_1}},$$

где Ω_{d_k} - область изменения k -го параметра актора.

Аналогично, по каждому из вычисляемых параметров, в паспорте актора хранится информация о диапазонах их изменений. Например, если в программе вычисляется значение кпд узла некоторого технического устройства, диапазоном этого параметра, очевидно, является отрезок $[0,1]$.

Введем событие S_E , связанное с возникновением ситуации ошибки (ERROR) при некотором сочетании исходных данных в процессе вычисления актора A_k .

Будем говорить о *надежной работе актора* A_k , если для любого вектора исходных данных из области определения актора вероятность возникновения ошибочной ситуации равна нулю, т.е. $\forall D_k^{in} \in \Omega_{A_k} \{P(S_E) = 0\}$.

Подмножество $\Omega_{A_k}^E$ множества определения актора будем называть *областью ошибок*, если $\forall D_k^{in} \in \Omega_{A_k}^E \subset \Omega_{A_k} \{P(S_E) = 1\}$.

Задача тестирования актора заключается либо в обнаружении хотя бы одного сочетания исходных данных из области ошибок актора, либо в установлении факта, что $\Omega_{A_k}^E = \emptyset$.

3.1.1 Метод независимых испытаний

Ошибочные ситуации в программных модулях возникают по разным причинам. По признаку воздействия на вычислительный процесс их можно разбить на три группы:

1). *Методические ошибки*, связанные с несоответствием результатов вычислений, полученных с помощью актора, реальным зависимостям, которые он описывает.

2). *Семантические ошибки*, связанные с “выпадением” значений выходных параметров акторов из областей значений, к которым они “приписаны”

3). *Фатальные ошибки*, связанные с прерыванием вычислительного процесса программного модуля.

Обнаружить и устранить ошибки первого типа чрезвычайно сложно. Проблема тестирования методических ошибок непосредственно примыкает к проблеме оценки адекватности ПП, где критерием истинности, как известно, является практика.

Что же касается тестирования семантических и фатальных ошибок, то отдельные элементы автоматизации поиска и устранения ошибок здесь допустимы.

Наиболее "вредными" ошибками следует считать фатальные ошибки, поскольку любая фатальная ошибка, возникшая даже во второстепенном модуле, может прервать сложный вычислительный эксперимент. В то время как последствия семантических или методических ошибок могут сказаться лишь на части полученных результатов или быть скорректированы соответствующими поправками.

В работе [1] приведено описание наиболее распространенных причин возникновения ошибочных ситуаций:

1. *Функциональное несоответствие программы алгоритму;*
2. *Неправильный тип, структура или значение переменной или константы.*
- Отсутствие начальной установки переменных;*
3. *Наложение массивов или переменных;*
4. *Неправильное формирование логического выражения для условного оператора;*
5. *Некорректное внесение изменений в программу.*

Некоторые из вышеперечисленных ошибок (2, 4, 5) могут быть выявлены на этапе синтаксического анализа (компиляции) модуля, некоторые (1, 4) влияют только на ход выполнения алгоритма, но не всегда приводят к аварийным ситуациям. Однако существуют ошибки, которые способны вызвать фатальный сбой в системе и привести к зависанию программы или даже потере полученных результатов.

Природа таких ошибок, как правило, имеет "арифметический" характер. Круг арифметических ошибок можно условно разделить на четыре основные группы:

1. *Ошибки деления на ноль*, или, так называемые, ошибки переполнения порядка. Данные ошибки возникают в тех случаях, когда в результате арифметических операций (обычно операций деления) получаются числа $f = f_m \cdot 10^e$, $e > \max_e$, где f_m — мантисса числа, e — порядок, $\max_e$ — максимально допустимый порядок при реализации конкретного типа данных.

2. *Ошибки потери порядка*. Возникают, когда образуются числа $f = f_m \cdot 10^e$, $e < \min_e$, где $\min_e$ — минимальный порядок для конкретного типа данных.

3. *Ошибки математических функций*. При вычислении арифметических функций может возникнуть ситуация, когда входной параметр находится за пределами области определения функции. Например, выполнение функции

нахождения натурального логарифма $\ln(x)$ приведет к арифметической ошибке при $x < 0$.

4. *Ошибки заикливания.* Заикливание программных модулей может происходить по нескольким причинам. Это может быть плохо сходящийся итерационный процесс, или неправильно реализованный бесконечный цикл с некорректным условием выхода. Причина может быть и более "низкого" уровня, когда начинает сказываться ограниченность программных или аппаратных ресурсов конкретной реализующей ЭВМ. Так или иначе, заикливание программного модуля, независимо от причины, может привести к аварийному завершению работы всей программы, поэтому этот фактор также необходимо учитывать при тестировании.

Известно несколько вариантов, позволяющих автоматизировать процесс определения заикливания программного модуля. Один из вариантов основан на задании максимального времени выполнения программы $T_{\max}$. При тестировании программного модуля по данной схеме, в фоновом режиме производится постоянный контроль за временем его исполнения, и если время работы модуля T_{mod} превышает наперед заданную величину $T_{\max}$, то можно считать, что модуль заиклился.

Если исходить из позиций проявления ошибочных ситуаций в процессе выполнения программных модулей, семантические и фатальные ошибки можно разбить на следующие три группы:

- *грубые ошибки* - проявляющиеся практически при любом запуске программы, при этом весьма высока вероятность обнаружения ошибки любым методом тестирования;

- *неточности* ошибки, появляющиеся не слишком часто и проявляющиеся в виде неустойчивой работы программы в зависимости от исходных данных. Вероятность обнаружения таких ошибок достаточна велика;

- *тонкие или редкие ошибки* - практически не проявляются в процессе исполнения программных модулей. Пользователь обычно считает, что работает с отлаженной программой. Вероятность обнаружения таких ошибок очень мала.

Грубые ошибки имеют область значений, соизмеримую с областью значений функции, и их идентификация не вызывает затруднений.

Хуже обстоит дело с тонкими ошибками, когда ошибочная ситуация теоретически реализуется в единственной точке. Причины возникновения таких ошибок можно условно разделить на два типа: логического и арифметического характера. Логический способ порождения тонкой ошибки позволяет получить ошибку любого уровня "тонкости". Например, выражение

$$\text{if } x = 1.0 \text{ then } y = 1.0 / (1.0 - x);$$

вызовет возникновение ошибочной ситуации в единственной точке: $x = 1.0$.

Идентификация такого рода логических ошибок практически невозможна. Однако их можно выявить другими методами, например, методом доказательства корректности программы [43].

При разработке программных продуктов в рамках технологии ГСП следует, по возможности, избегать создания базовых модулей со сложной логической структурой, реализуя лишь арифметические вычисления. Организация логических связей при этом перекладывается на агрегаты технологии ГСП.

Для обнаружения семантических и фатальных ошибок в *вычислительных модулях* предлагается использовать метод, напоминающий схему независимых испытаний Бернулли.

Сущность метода заключается в следующем. Пусть $\xi = (\xi_1, \xi_2, \dots, \xi_n)$ случайный вектор исходных данных актора A_k , равномерно распределенный в области определения актора $\xi \in \Omega_{A_k}$. Разыгрывая достаточно большое число независимых векторов ξ , при условии $\Omega_{A_k}^E \neq \emptyset$, мы с определенной вероятностью попадем в область $\Omega_{A_k}^E$ и тем самым обнаружим ошибку в программе. На рисунке 3.1 представлена функциональная схема алгоритма стохастического тестирования акторов ПОП. С помощью генератора псевдослучайных чисел из области определения актора (или ее подмножества) разыгрываются различные сочетания исходных данных актора. Описание области Ω_{A_k} "берется" из паспортных данных актора, хранящихся в базе данных предметной области программирования. В каждой точке ξ вычисляются выходные параметры актора, которые подвергаются семантическому анализу на принадлежность своим областям значений. По результатам анализа формируется область ошибочных данных $\Omega_{A_k}^E$, которая сохраняется в БД ПОП. Для ликвидации последствий фатальных ошибок, которые возникают вследствие некорректных арифметических операций над целыми и действительными числами, использовался механизм перехвата системных сообщений об ошибках, позволяющий разрешать конфликтные ситуации без аварийного завершения работы программы.

Перед началом процесса тестирования программного модуля происходит переустановка стандартных действий операционной системы на случай возникновения математических ошибок (например, ошибка "деления на ноль"). С помощью этого, при функциональном выполнении модуля, в случае возникновения прерывания, процесс не прекращается, а передача управления происходит к подсистеме ликвидации последствий математических ошибок. В частности, одними из действий, производимых этой системой, является очистка математического стека, сброс регистров сопроцессора и т.п.



Рис.3.1 Функциональная схема алгоритма тестирования акторов

Вероятность обнаружения ошибки зависит от соотношений “объемов”

области ошибок и области определения актора $P_E = \frac{V(\Omega_{A_k}^E)}{V(\Omega_{A_k})}$ (см. рис.3.2). Для

грубых ошибок и ошибок типа неточностей область ошибок составляет десятки и более процентов от области определения актора, и, следовательно, обнаружение таких ошибок не вызывает затруднений. Для оценки уровней вероятностей обнаружения ошибок перечисленных выше типов P_E , было предпринято небольшое исследование [34]. Статистическому анализу были подвергнуты два типа программ, задание на разработку которых было предложено семи программистам, имеющим разную квалификацию и различный практический опыт. При этом отслеживалась вся технологическая цепочка отладки программы. Исследования показали, что для грубых ошибок

P_E находится в пределах 0.8 ... 1.0; неточности (ошибки среднего уровня) определяют диапазон 0.02 ... 0.5; что же касается тонких ошибок, то вероятности их обнаружения $\ll 0.01$.

Исследование редких ошибок проводилось на тестовых функциях, имитирующей ошибочную ситуацию в единственной точке с координатами $x=1$, $y=1$ в квадрате $\Pi=[0.1; 2] \times [0.1; 2]$. В качестве таковой можно взять функцию

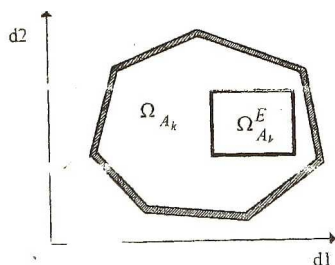


Рис.3.2

$f(x,y) = 1 / ((x-1)^2 + (y-1)^2)$, которая выполняет шесть арифметических операций и в точке (1,1) и порождает фатальную ошибку деления на ноль. Вероятность обнаружения ошибки в случае 15-ти разрядной арифметики составляет величину $> 1.0 \cdot 10^{-30}$, из чего следует вывод, что по результатам статистических испытаний практически невозможно обнаружить такие ошибки. С другой стороны, ограниченность разрядной сетки ЭВМ, распространение ошибок арифметических операций способны увеличить размеры области ошибочных ситуаций до приемлемых для тестирования размеров. Для проверки этой гипотезы было предпринято специальное исследование на модельной функции:

$$f(x,y) = 1 / ((x^m - C_m^1 x^{m-1} + \dots + (-1)^m C_m^m x^0) + (y^m - C_m^1 y^{m-1} + \dots + (-1)^m C_m^m y^0)). \quad (3.1)$$

С точки зрения вычисляемой функции, она эквивалентна предыдущей модели, однако количество арифметических операций в ней значительно больше. При $m=4$, она реализует 54 арифметические операции, а при $m=40$ около 440 и т.д. Вычислительные эксперименты подтвердили выдвинутую гипотезу. Уже начиная с $m=3$, область ошибок заметно расширилась, при этом вероятность обнаружения ошибки составила $P_E = 1.10^{-10}$, а при $m=5$ - $P_E = 1.10^{-6}$, что вполне доступно для алгоритма статистического испытаний. На рисунке 3.3 представлены результаты тестирования актора, функционирующего в соответствии с моделью (3.1). Здесь $D(\Omega_E)$ - диаметр области ошибок.

Полученный феномен можно объяснить следующими соображениями.

Пусть $x = x_t 10^t + x_{t-1} 10^{t-1} + \dots + x_0$ и $a = a_t 10^t + a_{t-1} 10^{t-1} + \dots + a_0$ представление десятичных чисел в t -разрядной арифметике. Рассмотрим задачу нахождения корней уравнения $(x-a)^2 = 0$ на ЭВМ с t -значной арифметикой. Выражение $(x-a)^2$ можно записать в виде:

$$\begin{aligned} (x-a)^2 &= (x_t - a_t)^2 10^{2t} + 2(x_t - a_t)(x_{t-1} - a_{t-1}) 10^{2t-1} + \\ &+ [(x_{t-1} - a_{t-1})^2 + (x_t - a_t)(x_{t-2} - a_{t-2})] 10^{2t-2} + \\ &+ 2[(x_t - a_t)(x_{t-3} - a_{t-3}) + (x_{t-1} - a_{t-1})(x_{t-2} - a_{t-2})] 10^{2t-3} + \\ &+ \dots + 2(x_1 - a_1)(x_0 - a_0) 10 + (x_0 - a_0)^2. \end{aligned}$$

Тогда исходное уравнение, с учетом ограниченности разрядной сетки (при условии, что округление результата производится путем отбрасывания младших разрядов), можно заменить системой уравнений:

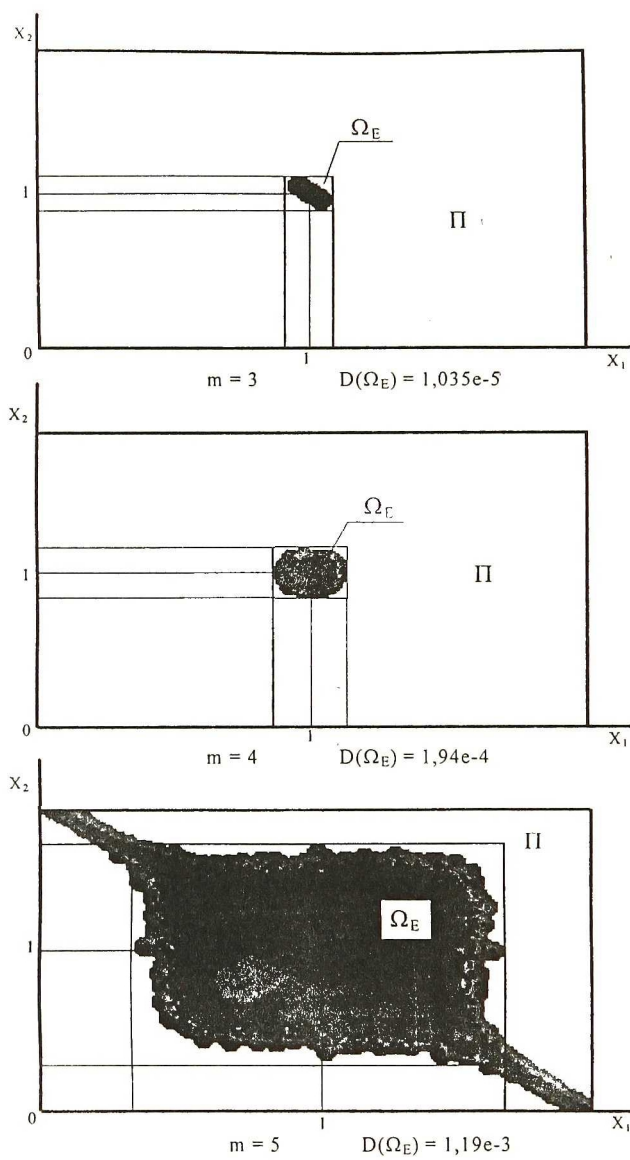


Рис. 3.3.

$$\begin{cases} (x_t - a_t)^2 = 0, \\ (x_t - a_t)(x_{t-1} - a_{t-1}) = 0, \\ (x_{t-1} - a_{t-1})^2 + 2(x_t - a_t)(x_{t-2} - a_{t-2}) = 0, \\ \dots \\ (x_t - a_t)(x_0 - a_0) + (x_{t-1} - a_{t-1})(x_1 - a_1) = 0 \end{cases} \quad (3.2)$$

Например при $t=4$ система (3.2) примет вид:

$$\begin{cases} (x_4 - a_4)^2 = 0, \\ (x_4 - a_4)(x_3 - a_3) = 0, \\ (x_3 - a_3)^2 + 2(x_4 - a_4)(x_2 - a_2) = 0, \\ (x_4 - a_4)(x_1 - a_1) + (x_3 - a_3)(x_2 - a_2) = 0, \\ (x_2 - a_2)^2 + 2(x_4 - a_4)(x_0 - a_0) = 0. \end{cases} \quad (3.3)$$

Корнями системы уравнений (3.3) являются:

$$\begin{cases} x_4 = a_4, \\ x_3 = a_3, \\ x_2 = a_2, \\ \forall x_1, \\ \forall x_0. \end{cases} \quad (3.4)$$

Однако, в реальных вычислениях на ЭВМ, учитывающих последовательность выполнения арифметических операций и операций округления чисел, реализуются не все корни (3.4). В то же время, как показывают вычислительные эксперименты, их достаточно большое количество.

Полученные результаты можно существенно усилить, если в процессе вычислений искусственно ограничить размеры разрядной сетки ЭВМ.

Пусть t - размер разрядной сетки гипотетической ЭВМ, r - область (в младших разрядах $r \in [0, t]$) "нечувствительности" к арифметическим операциям "+", "-". Переопределим арифметические операции следующим образом:

$$a \pm b = \begin{cases} 0, & |[a]_t \pm [b]_t| < \frac{\max\{[a]_t, |[b]_t|\}}{10^{t-r}} \\ [a]_t \pm [b]_t, & \text{иначе} \end{cases} \quad (3.5)$$

$$a * b = [[a]_t * [b]_t]_t. \quad (3.6)$$

$$a / b = [[a]_t / [b]_t]_t. \quad (3.7)$$

На рисунке 3.4 представлены зависимости изменения диаметра области ошибок $D(\Omega_E)$ от размера разрядной сетки t . Вычислительные эксперименты производились над моделями:

- D1: $f(x, y) = 1 / [(x - 0.5)^2 + (y - 0.5)^2]$;
- D2: $f(x, y) = 1 / [(x^2 - x + 0.25) + (y^2 - y + 0.25)]$;
- D3: $f(x, y) = 1 / [(x^3 - 1.5x^2 + 0.75x - 0.125) + (y^3 - 1.5y^2 + 0.75y - 0.125)]$.

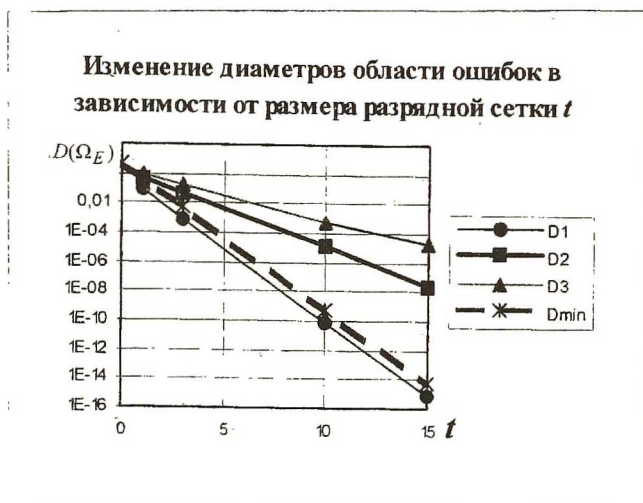


Рис. 3.4

Полученные результаты показывают, что с уменьшением разрядной сетки диаметр области ошибок нарастает как показательная функция по основанию 10 ($D(t) = \beta 10^{-\alpha \cdot t}$). Причем чем больше арифметических операций, тем больше диаметр области ошибок. Для сравнения на рисунке 3.4 (линия D_{min}) показано изменение величины относительной погрешности $D_{min} = 5 \cdot 10^{-t}$, что соответствует диаметру области ошибок с минимальным количеством операций. Как видно из рисунка, изменение диаметра области ошибок для модели D1 практически совпадает с графиком D_{min} . Область ошибок можно искусственно увеличить если при выполнении операций сложения или вычитания в случаях когда результат вычисления составляет 10, 20 и более процентов от суммы модулей используемых чисел, результат приравнять нулю (см. формулу (3.5)). На рисунке 3.5 показаны зависимости диаметров области ошибок от размерности разрядной сетки ЭВМ t , когда γ составляет 30% от t (линия D4) и 50% от t (линия D5).

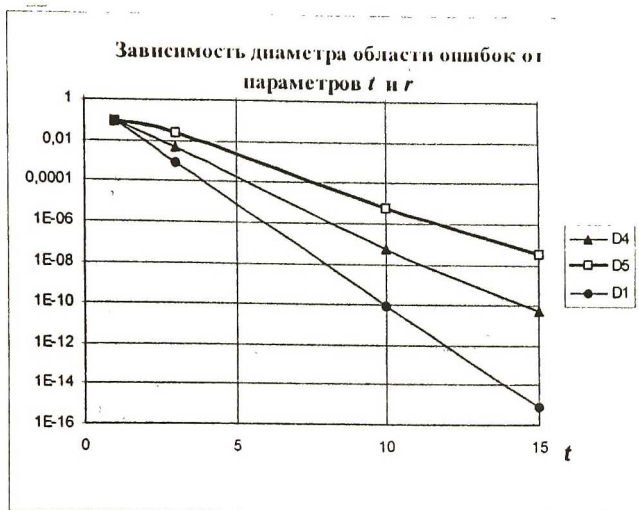


Рис. 3.5

Полученные результаты позволяют предложить ускоренный метод тестирования акторов. Идея метода основывается на искусственном переводе ошибок из категории *редких* ошибок в категорию *грубых* или *средних* за счет “огрубления” инструмента вычислений.

На первом этапе тестирования актора все арифметические операции переопределяются для одно или двух разрядной сетки ($t=1$ или $t=2$). Если $\Omega_E = \emptyset$, то это говорит либо об отсутствии ошибок в программе, либо о том, что вероятность ошибки $P_E < 1.0 \cdot 10^{-30}$ (для единичного квадрата и датчика случайных чисел с периодом $M=2000000$).

Если ошибка при уровне $t=1$ будет обнаружена, то в этом случае необходимо оценить размеры области Ω_E , сжать пространство поиска Π до размера Ω_E (или меньшего размера), увеличить параметр t и повторить тестирование.

Процесс продолжается до тех пор пока t не сравняется с истинным значением разрядной сетки ЭВМ.

Обсудим еще один важный вопрос, связанный с выбором датчика случайных чисел для алгоритма тестирования модулей. Формально для этих целей пригоден любой датчик псевдо-случайных чисел, имеющий достаточно большой период. Однако для целей тестирования модулей наиболее важной характеристикой датчика является обеспечение одинаковой плотности просмотра любого участка области тестирования. Таким требованиям удовлетворяют Π_r последовательности. Метод генерации равномерно распределенных векторов, предложенный И.М. Соболев и Р.Б. Статниковым,

гарантирует попадание в любой двоичный участок куба K^n не менее чем $2^{\tau+1}$ точек.

3.1.2 Адаптивный алгоритм тестирования акторов

Предлагаемый способ тестирования объектов ГСП и базисных функций основывается на принципе генерации в области определения функции равномерно распределенных случайных векторов. Эффективность поиска ошибок в программах в данном случае зависит от соотношения “объема” критической гиперобласти ошибок Ω_E (области данных из которой вызываются ситуации программных прерываний) и “объема” пространства тестирования Π . Для “тонких” ошибок, когда $V_{\Omega_E} \approx \delta^n = (10^{-t+1})^n$ вероятность проявления ошибки $p_E = V_{\Omega_E} / V_{\Pi}$ в процессе тестирования функции представляет собой малую величину, а обнаружение ошибки становится проблематичным делом. Здесь, δ — относительная погрешность представления чисел в ЭВМ, n — размерность пространства поиска, t — разрядность ЭВМ.

Ситуация осложняется, если учесть влияние размерности пространства поиска n . Рассмотрим гиперкуб, в который вписан гипершар. Объем n -мерного гипершара представим формулой $V_{\text{ш}}^{(n)} = A_n R^n$, где A_n — объемный коэффициент, R — радиус гипершара. Тогда вероятность попадания в гипершар равномерно распределенного случайного вектора оценивается величиной $p = A_n / 2^n$, т.е. p пропорциональна величине объемного коэффициента. Можно показать, что для нечетных n объемный коэффициент вычисляется по формуле:

$$A_{2m+1} = 2(2\pi)^m / (2m+1)!! \quad (3.8)$$

Числовая последовательность (3.8) является строго убывающей и ограниченной снизу, следовательно, она имеет предел. Используя определение предела числовой последовательности несложно показать, что $\lim_{m \rightarrow \infty} A_{2m+1} \rightarrow 0$, но тогда $p \rightarrow 0$. Из чего следует вывод, что при определенных значениях n невозможно обнаружить не только “тонкие” программные ошибки, но и обнаружить гипершар, вписанный в гиперкуб. На рисунке 3.6 показаны зависимости от n объемного коэффициента и вероятность обнаружения гипершара. Из рисунка видно, что рациональное значение верхней границы размерности пространства поиска равно 8, когда значение объемного коэффициента A_n еще находится в пределах своего максимума. Дальнейшее увеличение n приводит к непроизводительным затратам машинного времени в силу малости p .

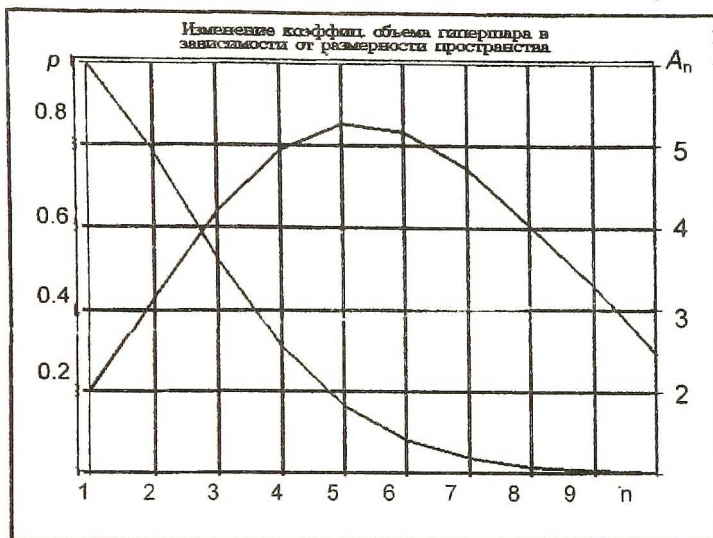


Рис. 3.6

Очевидно, что любые мероприятия, связанные с увеличением вероятности p_E , приводят к более уверенному поиску программных ошибок. Одним из методов увеличения p_E является уменьшение области просмотра, т.е. уменьшение V_{Π}

Предлагается адаптивный алгоритм случайного поиска, основу которого составляют следующие идеи:

1. Вместо глобального поиска реализуется локальный поиск в небольшом гиперкубе, постепенно перемещающемся в пространстве поиска в направлении подозрительных точек.

2. Центр гиперкуба постоянно перемещается в “наилучшие” среди найденных точек.

3. Размеры гиперкуба изменяются (адаптируются) в зависимости от локальных свойств исследуемой функции.

Относительно исследуемой функции $y = f(x_1, x_2, \dots, x_n)$ предполагается, что она является кусочно-непрерывной функцией с конечным или счетным числом разрывов первого или второго рода.

Адаптация размеров локального гиперкуба σ производится из следующих соображений.

Пусть через Ω_- обозначено подмножество точек гиперкуба, для которых выполняются условия:

$$\Omega_- = \{X | f(X) < f(X_0)\}.$$

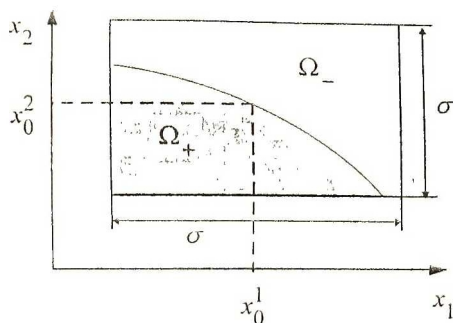


Рис. 3.7

Соответственно через Ω_+ обозначим множество точек гиперкуба, в которых функция принимает большее или равное значение по сравнению с $f(X_0)$ - значением функции в центре гиперкуба (см. рисунок 3.7). Через p обозначим вероятность события $A = \{\xi \in \Omega_+\}$, где ξ - равномерно распределенный в гиперкубе вектор.

Пусть $\nu_p(N)$ - число появлений события A в N независимых испытаниях. Тогда вероятность улучшения значения $f(X_0)$ (поиск ошибок связанных с переполнением порядка) в N испытаниях с доверительной вероятностью δ можно вычислить из уравнения:

$$P\{\nu_p(N) \geq 1\} = \delta \text{ или}$$

$$P\{\nu_p(N) \geq 1\} = \sum_{k=1}^N C_N^k p^k (1-p)^{N-k} = 1 - P\{\nu_p(N) = 0\} = 1 - (1-p)^N = \delta,$$

или

$$N = \frac{\ln(1-\delta)}{\ln(1-p)}. \quad (3.9)$$

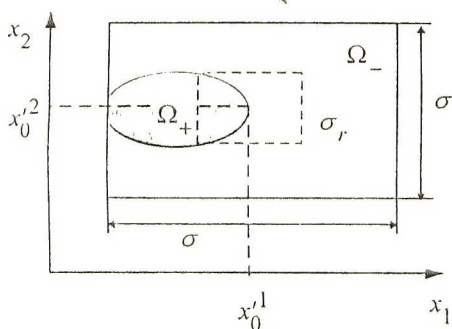


Рис. 3.8

Формула (3.9) определяет количество "пробных" точек, улучшающих значение функции в центре гиперкуба X_0 с доверительной вероятностью δ . Если прогноз не оправдался и реальное число испытаний N оказалось значительно больше, чем прогнозируемое N_0 (вычисленное по формуле (3.9)), то последнее можно объяснить изменением топологии исследуемой функции в положении нового центра

гиперкуба X_0' после перемещения его в "лучшую" точку (см рис. 3.8). Для того чтобы добиться заданных темпов "улучшения" функции в новых условиях,

следует уменьшить пространство поиска от σ до σ_r (см. рис.3.7). Пропорции сжатия диаметра гиперкуба определим из следующих соотношений.

Из уравнения (3.3) определим прогнозируемый уровень вероятности успеха:

$$p_0 = 1 - (1 - \delta)^{1/N_0} = \frac{V_{\Omega_+}}{\sigma^2}.$$

Реальное количество испытаний определяет иной уровень вероятности:

$$\tilde{p} = \tilde{V}_{\Omega_+} / \sigma^2 \quad \text{или} \quad \tilde{V}_{\Omega_+} = \sigma^2 (1 - (1 - \delta)^{1/N_r}).$$

Из соображений подобия изменим диаметр в следующих пропорциях:

$$p_0 = \tilde{p} = \frac{\tilde{V}_{\Omega_+}}{\sigma^2} = \frac{\tilde{V}_{\Omega_+}}{\tilde{\sigma}^2} \quad \text{или}$$

$$\tilde{\sigma}^2 (1 - (1 - \delta)^{1/N_0}) = \sigma^2 (1 - (1 - \delta)^{1/N_r}).$$

В итоге имеем формулу изменения пропорций диаметра гиперкуба, которая пригодна как для "сжатия" гиперкуба, так для его расширения:

$$\tilde{\sigma} = \left(\sigma^n \frac{1 - N_r \sqrt{1 - \delta}}{1 - N_0 \sqrt{1 - \delta}} \right)^{\frac{1}{n}}. \quad (3.10)$$

Формула (3.10) представлена для случая n-мерного пространства. На рисунке 3.9 показана зависимость изменения диаметра гиперкуба от

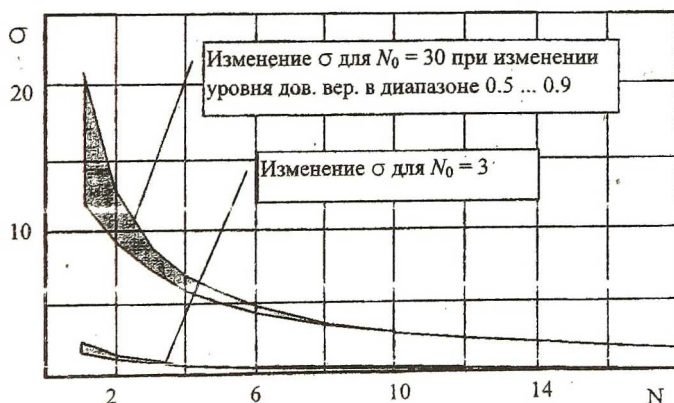


Рис. 3.9

наблюдаемого числа испытаний N_T до достижения “успеха” при заданном прогнозе, оцениваемом N_0 . Как видно из рисунка, $\tilde{\sigma}$ практически не зависит от уровня доверительной вероятности, но существенно изменяется при изменении N_0 . Значению параметра $N_0=3$ соответствует вероятность $p=0.5$, что характерно для оптимизируемых параметров удаленных на значительное расстояние от экстремума функции. В этом случае для увеличения скорости сходимости следует увеличить диаметр гиперкуба, используя формулу (3.10). Темпы увеличения диаметра будут еще больше, если при этом увеличить значение N_0 .

В окрестностях экстремума функции топология такова, что $p \rightarrow 0$. В связи с чем, для увеличения скорости сходимости целесообразно уменьшать величину N_0 .

Адаптивное тестирование модулей

В тех случаях, когда акторы реализуют функциональные зависимости одних переменных от других (вычислительные модули), появляется возможность использования АСП для идентификации ошибочных ситуаций в программе. Идея применения АСП для тестирования модулей основывается на том, что ошибки типа деления на ноль (тонкие ошибки) возникают у функции $f(X)$ в точках (линиях) разрыва второго рода, для функции $1/f(X) \rightarrow \pm\infty$. Из чего следует, что поиск ошибочных ситуаций можно заменить поиском абсолютного минимума или максимума для функции $1/f(X)$.

Вычислительные эксперименты над моделью (3.1) показали, что ошибочная ситуация возникает через 1500-3600 обращений к функции, практически из любой начальной точки области ее определения.

В работе [38] Г. Майерс для проверки квалификации составления тестов программистом предлагает написать тесты для программы, которая вычисляет корни квадратного уравнения $AX^2 + BX + C = 0$. Данный пример иллюстрирует трудности при составлении высококачественных тестов программ. Грюнбергером были разработаны семь тестов, проверяющие ситуации, возникающие при различных сочетаниях параметров A, B , и C , например, когда корни действительные и разные, когда имеются кратные корни, когда дискриминант меньше нуля, когда $A=0$ и т.д.

С помощью метода статистического тестирования эта задача решается при построении “фотографии” работоспособности модуля (см. рис.3.10).

Из рисунка видно, что область ошибочных ситуаций возникает по двум причинам: деление на ноль, когда $A=0$ и в случае извлечения квадратного корня из отрицательного числа, когда дискриминант уравнения меньше нуля.

Для большей наглядности представляемых результатов, в данном примере коэффициент C был принят равным единице.

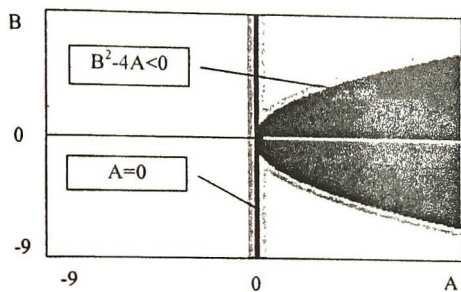


Рис.3.10

Поиск велся в квадрате $10^{12} \times 10^{12}$.

Сложнее оказалось идентифицировать ситуацию *деление на ноль* ($A=0$).

Дело в том, что ноль в знаменателе формулы $X = \frac{-B \pm \sqrt{B^2 - 4A}}{2A}$ формируется в "чистом" виде (без участия арифметических операций сложения). В таких условиях даже на ЭВМ ошибочная ситуация возникнет, когда выполнится условие $|A| < 1.7 \cdot 10^{-308}$. Тем не менее, данная ситуация была обнаружена методом АСП за 32000 итераций, при стартовой точке $A=-80000$, $B=80000$. Данный пример поучителен, поскольку подсказывает идею в процессе тестирования модулей в многомерном пространстве в обязательном порядке просматривать координатные плоскости и (или) координатные оси.

3.2 Тестирование агрегатов технологии ГСП

При тестировании агрегата в технологии ГСП предполагается, что все объекты, из которых он составлен (акторы или агрегаты), прошли автономное тестирование и получили "сертификат качества". Это означает, что каждый *актор* с заданной доверительной вероятностью не содержит ошибок в пределах своей области определения, а каждый *агрегат* прошел тесты на логическую целостность. Следовательно, тестирование агрегата как функциональной зависимости методами аналогичными тем, которые используются для тестирования акторов, *нецелесообразно*. Действительно, если в логике агрегата не допущены ошибки, то верная последовательность надежных модулей должна давать правильный результат. На данном этапе разработки программы ошибки возникают либо за счет неверно составленных или употребленных логических выражений (предикатов), либо за счет неверной организации управления, т.е. неправильной схемы (графа) программы.

Эксперименты с представленной моделью показали, что методом статистических испытаний (МСИ) ошибка обнаруживалась, если размеры области поиска ограничивались рамкой $10^6 \times 10^6$ (при $t=1$), в то время как АСП обнаружил ошибочную ситуацию (корень квадратный из отрицательного числа) за 1332 итераций, при этом были получена ошибочная точка $A=13785203$, $B=-3377811$.

Ошибки неверного употребления данных или организации межмодульного информационного интерфейса в технологии ГСП маловероятны. Во-первых, неправильное употребление данных должно быть установлено еще на этапе тестирования акторов. Во-вторых, при автоматической организации межмодульного информационного интерфейса ошибки такого рода исключены.

Из сказанного выше следует, что при тестировании агрегатов необходимо проверять корректность графа управления вычислительным процессом, а также полноту и непротиворечивость используемых предикатов.

3.2.1. Проверка корректности организации ветвлений из вершин графа

Логика управления на графе агрегата определяется используемой в нем системой предикатов. При составлении и применении предикатов могут быть допущены следующие ошибки:

1. Некорректное описание предиката, когда предикат является либо тождественно истинной функцией, либо тождественно ложной.

2. Множества интерпретаций истинности предикатов, “исходящих” из вершины графа, не покрывают область определения данных, на которых они заданы (полнота системы предикатов, исходящих из вершины графа).

3. Множества интерпретаций истинности предикатов, реализующие ветвление, имеют общие области данных (нестрогое разделение области определения данных).

Ошибки первого типа встречаются достаточно редко и могут быть обнаружены формальными методами. Ошибки второго и третьего типа являются наиболее типичными и связаны с организацией мыслительных процессов у человека. Несоответствие образной, зачастую подсознательной, схемы логических выводов человека формально-логическому способу их представления, принятому в языках программирования, порождает огромное количество неточностей при реализации “задуманного” логического условия и его воплощения в кодах языка программирования.

Область интерпретации системы предикатов, исходящих из вершины графа, может быть исследована “сканированием” этой области методом статистических испытаний, который применялся для тестирования акторов (см. п.3.1).

Действительно, система предикатов, связанных с вершиной графа, с формальной точки зрения реализует высказывание

$$\forall x_1, \forall x_2, \dots, \forall x_m (P_1(x_1, \dots, x_{n_1}) \vee P_2(x_1, \dots, x_{n_2}) \vee \dots \vee P_m(x_1, \dots, x_{n_m})). \quad (3.11)$$

Формула (3.11) имеет сколемовскую стандартную форму, т.е. является $\forall$ -формулой [5], но тогда область интерпретации формулы (3.11) совпадает с областью интерпретации дизъюнкта

$$D = P_1(x_1, \dots, x_{n_1}) \vee P_2(x_1, \dots, x_{n_2}) \vee \dots \vee P_m(x_1, \dots, x_{n_m}),$$

и изучение корректности системы предикатов $\{P_1, \dots, P_m\}$ связано с исследованием свойств дизъюнкта D.

Например, исследование наличия точек в области интерпретации системы предикатов, когда все предикаты одновременно ложны (тупиковая ситуация), может быть проведено с помощью функции

$$\varphi(X) = \begin{cases} 1/0 & \{X | P_1(X) \vee \dots \vee P_m(X) = 1\} \\ 1 & \text{иначе} \end{cases} \quad (3.12)$$

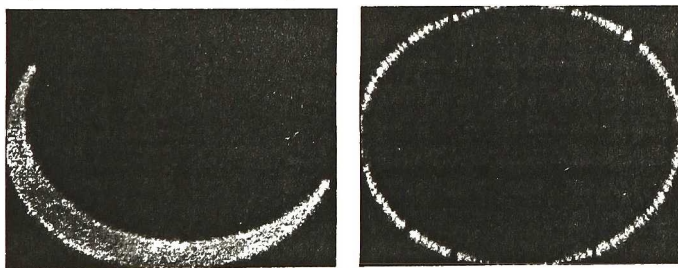
Функция $\varphi(X)$ моделирует ситуацию деления на ноль, если все предикаты ложны одновременно в некоторой точке. Теперь с помощью алгоритма статистических испытаний, используя функцию (3.12), можно “сфотографировать” область интерпретации системы предикатов, исходящих из вершины графа.

На рисунке 3.11 представлены “портреты” предикатов

$$P_1(x, y) = ((x - 1)^2 + (y - 1)^2 \leq 1),$$

$$P_2(x, y) = (|x - 1| < 1) \& (|y - 1| < 1) \& ((x - 1)^2 + (y - 1)^2 > 1),$$

разделяющих квадрат на следующие два подмножества: круг единичного радиуса и квадрат с вырезанным кругом. В первом случае была допущена ошибка несогласования центров окружностей в предикатах P_1 и P_2 . Во втором случае, пропущен знак равенства при описании окружности предиката P_1 .



а)

б)

Рис. 3.11

Аналогично можно построить функции для тестирования ситуации наличия общих областей истинной интерпретации предикатов, исходящих из вершины графа.

Однако если область интерпретации предикатов достаточно велика, то “разрешающая” способность алгоритма статистических испытаний может оказаться недостаточной для выявления ошибочных ситуаций. С другой стороны, применение алгоритма случайного поиска в данном случае

невозможно из-за того, что предикаты дают лишь качественную оценку для любых точек фазового пространства.

Ситуация изменится, если воспользоваться “решающей” функцией [28].

В технологии ГСП используется исчисление предикатов первого порядка с сигнатурой, включающей отношения порядка. Это означает, что при построении предикатов используются алгебраические выражения, содержащие отношения порядка ($<$, $>$, $=$, $><$, $<=$, $>=$), т.е. произвольный предикат представляется следующим образом:

$$P(X) = f(X) \theta 1, \quad (3.13)$$

где θ - любое из перечисленных выше отношений порядка, а $f(X)$ - произвольная функция, непрерывная в области интерпретаций предиката. Значениям истинности формулы (3.13) можно поставить в соответствие множество:

$$M = \{X | f(X) \theta 1\}. \quad (3.14)$$

В качестве базисных отношений выберем $<=$ и $<$. Пусть заданы базисные множества $M_{\leq} = \{X | f(X) \leq 1\}$ и $M_{<} = \{X | f(X) < 1\}$. Введем функцию $\varphi(X) = -f(X) + 2$, через которую определим дополнения базисных множеств: $\overline{M_{\leq}} = \{X | \varphi(X) < 1\}$ и $\overline{M_{<}} = \{X | \varphi(X) \leq 1\}$.

Введем предикаты (составленные в сигнатуре $<=$, $<$) $P_{\leq}$, $P_{<}$, $\overline{P_{\leq}}$, $\overline{P_{<}}$, через которые можно выразить все остальные варианты использования отношений порядка (см. таблицу 3.1).

Таблица 3.1

Исходный предикат	Представление через базовые предикаты
$(f(X) > 1)$	$\overline{P_{\leq}}$
$(f(X) \geq 1)$	$\overline{P_{<}}$
$(f(X) = 1)$	$\overline{P_{<}} \& P_{\leq}$
$(f(X) \neq 1)$	$\overline{P_{\leq}} \vee P_{<}$

Таким образом, любое высказывание можно выразить через отношения порядка типа $<=$ и $<$. Будем говорить, что система предикатов приведена к стандартизированной форме, если при построении предикатов используются отношения $<=$ или $<$.

Между теоретико-множественными и логическими операциями существует связь, которая представлена в таблице 3.2.

Таблица 3.2.

Логика	Множества
$\overline{P(X)}$	$\overline{\{X f(X) \leq 1\}} = \{X \varphi(X) < 1\}$ или $\overline{\{X f(X) < 1\}} = \{X \varphi(X) \leq 1\}$
$P(X) \& R(X)$	$\{X f(X) \leq 1\} \cap \{X \varphi(X) \leq 1\}$
$P(X) \vee R(X)$	$\{X f(X) \leq 1\} \cup \{X \varphi(X) \leq 1\}$

Определим решающую функцию $F(X)$ следующим образом:

1. Для каждого предиката введем функции для операции $\leq$:

$$\beta_k(X) = \begin{cases} 1 & \text{если } f_k(X) > 1 \\ 0 & \text{если } f_k(X) \leq 1 \end{cases}$$

(фактически $\beta_k = \overline{P_k(X)}$) для операции $<$:

$$\beta_k(X) = \begin{cases} 1 & \text{если } f_k(X) \geq 1 \\ 0 & \text{если } f_k(X) < 1 \end{cases}$$

2. Для операции объединения двух множеств “решающую” функцию зададим следующим образом:

$$F(X) = (\beta_1(X)f_1(X) + \beta_2(X)f_2(X)) / ((\overline{\beta_1(X)} + \overline{\beta_2(X)} + \beta_1(X)\beta_2(X)) + \beta_1(X)\beta_2(X)(f_1(X) + f_2(X)) / 2 \quad (3.15)$$

3. Для операции пересечения двух множеств, имеем:

$$F(X) = (\beta_1(X)f_1(X) + \beta_2(X)f_2(X)) / (\beta_1(X) + \beta_2(X) + \overline{\beta_1(X)}\overline{\beta_2(X)} + \overline{\beta_1(X)}\overline{\beta_2(X)}(f_1(X) + f_2(X)) / 2 \quad (3.16)$$

Аналогично можно определить любые теоретико-множественные операции. Теперь результат объединения двух множеств через решающую функцию (3.15) определится формулой $\{X|F(X) \leq 1\}$. Однако важнее другое: решающая функция устанавливает количественные соотношения между точками области интерпретаций предикатов, связанных логическим “или”. Причем функция $F(X)$ убаывает в направлении истинности сложного высказывания. Следовательно, для нахождения параметров истинности сложного высказывания, можно применить алгоритм адаптивного случайного поиска, эффективность которого значительно выше, чем у метода статистических испытаний.



Рис. 3.12 Функциональная схема тестирования ветвлений вершин графа

На рисунке 3.12 представлена функциональная схема алгоритма контроля корректности организации ветвления из вершин графа. Подобные проверки в технологии ГСП реализуются в автоматическом режиме.

На первом этапе все логические высказывания приводятся к стандартизованному виду, когда в записи логических функций используются отношения $\leq$ и $<$. Далее, исследуемая формула преобразуется к минимальной дизъюнктивной нормальной форме, на основе которой строится минимальная нормальная форма Кантора [14], описывающая логическое выражение в рамках теории множеств. На основе нового выражения формируется "решающая" функция, минимизация которой позволяет установить факт корректности логического ветвления.

Если минимум "решающей" функции меньше единицы, то это означает, что существуют такие сочетания исходных данных, когда вершина графа не содержит дуг, помеченных значением истинности предикатов. Решающая функция минимизируется с помощью алгоритма АСП. В точке конфликта

производится “фотографирование” фрагмента области интерпретации системы предикатов для дальнейшего анализа причин ошибочной ситуации.

В технологии ГСП, за счет использования механизма назначения приоритетов дугам графа, не требуется жесткого разделения области определения системы предикатов на непересекающиеся подмножества истинности. Более того, если последний по приоритету предикат является тождественно истинным, то конфликтная ситуация вообще не наступает. Однако следует помнить, что логические ошибки, допущенные в более приоритетных предикатах, в этом случае будут “перенаправлены” на последний, тождественно истинный предикат и развитие вычислительного процесса может пойти по ложному направлению.

Во избежание подобных ситуаций следует рекомендовать *разрабатывать логические функции для всех дуг, исходящих из вершины графа.*

3.2.2 Тестирование структуры агрегата

Основой метода автоматизации тестирования структуры агрегата является разбиение графового образа программы на множество маршрутов, которые описывают пути реализации алгоритма.

Для построения множества маршрутов в технологии ГСП применяется метод частичного перебора вариантов с отбрасыванием заведомо ложных путей. Особенностью данного метода является то, что он позволяет строить цепочки для любых агрегатов, в том числе и для агрегатов, имеющих циклы. Проверка структурной корректности агрегата сводится к анализу всех его вычислительных маршрутов.

При анализе вычислительных маршрутов следует выделить две задачи:

- 1). Обнаружение логических ошибок формирования маршрута.
- 2). Проверка реализуемости вычислительного маршрута на исходных данных граф-агрегата.

Логические ошибки возникают, если при составлении предикатов, управляющих ходом вычислений на графе, допущены ошибки не “пропускающие” развитие вычислений по тестируемому маршруту.

Пусть $M_k = “0-1-2-3”$ тестируемый маршрут некоторого агрегата. Будем говорить, что маршрут M_k не содержит логических ошибок, если предикат

$$P_{M_k}(X) = (P_{01} \& \dots \& \overline{P_{0m_0}}) \& (P_{12} \& \dots \& \overline{P_{1m_1}}) \& \dots \& (P_{23}) \quad (3.17)$$

имеет интерпретацию истинности на множестве исходных данных маршрута M_k . Здесь $P_{ij}(X)$ - предикат, помечающий дугу, исходящую из вершины i в вершину j ; предикаты $\overline{P_{ik}}(X)$ помечают переходы, по которым не должно передаваться управление.

Для исследования предиката (3.17) можно применить алгоритм, который использовался при тестировании ветвлений вершин графа (см. п. 3.2.1).

Рассмотрим пример тестирования одного из маршрутов агрегата алгоритма адаптивного случайного поиска. На рисунке 3.13 представлена схема маршрута.

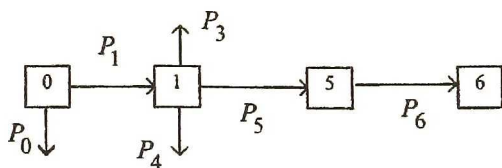


Рис. 3.13.

Предикаты маршрута имеют следующее содержание:

$$\begin{cases} P_1 = (x > y), \\ P_2 = .T., \\ P_3 = (x > 1.5yz) \& (y > 12), \\ P_4 = .T., \\ P_5 = (x > 1.8yz) \\ P_6 = .T. \end{cases} \quad (3.18)$$

Предикат маршрута имеет вид:

$$P_M = P_1 \& P_3 \& P_5. \quad (3.19)$$

Приведем предикаты к стандартизованному виду:

$$\begin{cases} P_1 = (x - y + 1 > 1) = \overline{(-(x - y + 1) + 2 \leq 1)} = \overline{A} \\ P_3 = \overline{(-(x - 1.5yz + 1) + 2 \leq 1)} \& \overline{(-(y - 11) + 2 \leq 1)} = \overline{B} \& \overline{C} \\ P_5 = \overline{(-(x - 1.8yz + 1) + 2 \leq 1)} = \overline{D} \end{cases} \quad (3.20)$$

Учитывая (3.14), формулу (3.13) можно представить в виде:

$$P_M = \overline{AD}(B \vee C).$$

Введем функции $f_A(X) = x - y + 1$, $f_B(X) = x - 1.5yz + 1$, $f_C(X) = y - 11$, $f_D(X) = x - 1.8yz + 1$. Множество интерпретаций истинности предиката P_M , тогда выглядит

$$\{X | P_M(X) = 1\} = \{X | \varphi_A(X) \leq 1\} \cap \{X | \varphi_D(X) \leq 1\} \cap (\{X | f_B(X) < 1\} \cup \{X | f_C(X) < 1\}), \quad (3.21)$$

где $\varphi_A(X) = -f_A(X) + 2$, $\varphi_D(X) = -f_D(X) + 2$.

На основе формулы (3.21), применяя несколько раз правила (3.15) и (3.16), строится “решающая” функция $F(X)$. Минимизация функции $F(X)$ адаптивным алгоритмом случайного поиска позволяет установить точку интерпретации истинности предиката $P_M(X)$ ($F(X) < 1$): $x=336$, $y=5$, $z=26$ за 35 обращений к “решающей” функции. С помощью алгоритма статистических испытаний были получены “фотографии” фрагмента области определения тестируемого маршрута (см. рис. 3.14).

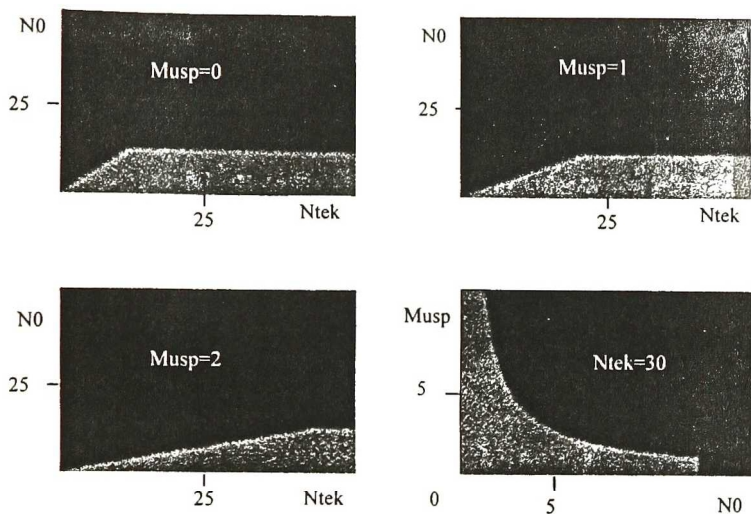


Рис. 3.14

Проведенные исследования позволяют установить “принципиальную” возможность развития вычислительного процесса по выбранному маршруту. В реальных вычислениях на ход вычислительного процесса существенное влияние окажут вычисления, производимые объектами маршрута, поскольку они могут изменять значения данных агрегата.

Для проверки реализуемости вычислительного маршрута на исходных данных граф-агрегата можно применить алгоритм статистических испытаний агрегата.

Для этого в БД предметной области регистрируются все вычислительные маршруты агрегата. Так, например, для агрегата “Пустой прогон”, являющегося одним из объектов программы адаптивного случайного, таких маршрутов будет 4 (см. рис 3.15): M1456, M156, M13, M26.

Для тестирования агрегата в данном случае используется специализированная граф-машина, выполняющая в том числе и сбор статистических данных по реализации тех или иных маршрутов. Автономное тестирование агрегата показало, что все выделенные маршруты реализовались в процессе испытаний. Тестирование производилось по переменным Ntek, N0, Musp, Sibaz, являющихся исходными данными для агрегата “Пустой прогон”. На рисунке 3.16 показаны относительные частоты исполнения маршрутов, зарегистрированные при тестировании агрегата, в процессе его испытаний, и при оптимизации функции Розенброка, в процессе его реальной работы.

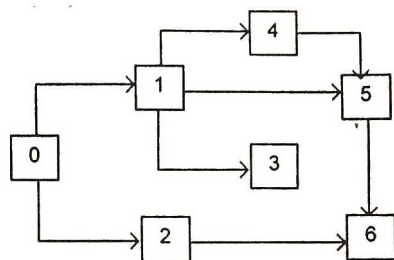


Рис. 3.15. Схема агрегата “Пустой прогон”

Полученные результаты тестирования агрегата “Пустой прогон”, показывают, что агрегат не содержит логических ошибок и все его маршруты алгоритмически реализуемы.

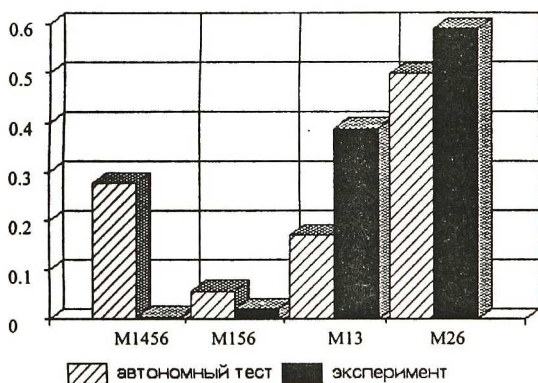


Рис 3.16.

Частотные характеристики реализации маршрутов в условиях опытной эксплуатации агрегата, как объекта процедуры адаптивного случайного поиска, показывают, что наиболее часто реализуется маршрут M26 (60%), который в программе обеспечивает сбор информации для определения размеров области поиска. Реже реализуются маршруты M1456 и M156 (1% и 4% соответственно), отвечающие за сжатие базового размера области поиска. Полученные данные соответствуют интуитивным представлениям разработчика программы.

3.3 Оценка надежности объектов

Надежность является важным и естественным требованием, предъявляемым к качеству разрабатываемого программного продукта. В области надежности аппаратуры достигнут уровень, позволяющий инженеру предсказать надежность его продукта. Теория надежности аппаратуры развита хорошо, поэтому при исследовании надежности программного обеспечения большинство методов заимствует идеи теории надежности технических устройств [12, 35, 38].

Однако, с точки зрения надежности, между техническими устройствами и программными продуктами, кроме несомненных аналогий, имеются существенные различия.

Как правило, плохо написанный программный модуль, как и плохо собранное устройство, "выходит из строя" на первых же этапах тестирования. Однако, на этом сходство заканчивается. Дело в том, что любое, даже самое надежное техническое устройство, подвержено эксплуатационному износу и со временем начинает отказывать из-за старения. Программные модули лишены подобных недостатков, и их надежность может со временем только увеличиваться за счет устранения выявляемых ошибок. Поэтому теоретически может возникнуть ситуация, когда при тестировании программного модуля ошибки уже не будут обнаруживаться.

Наиболее общее определение *надежности* ПП предлагает Б. Мейер [43], он определяет надежность как *способность программы давать разумные результаты во всех возможных окружениях и, в частности, в аномальных условиях*.

При разработке моделей надежности ПП, *надежность* чаще понимают в узком смысле как вероятность того, что ни одна программная ошибка не проявится в течение времени t .

Наиболее известными моделями надежности ПП, опирающимися на теорию надежности аппаратуры, являются модели Джелинского-Моранды, Шумана, Нельсона [38, 35, 12].

Большинство моделей надежности ПП предполагает своеобразный способ подсчета времени наработки на отказ программных продуктов. Например, в соответствии с моделью Шумана во временном интервале функции надежности учитываются лишь затраты времени на выявление ошибок с помощью тестов. При этом время исправного функционирования ПП не учитывается. Данный подход оправдан с позиции сведения задачи оценки надежности ПП к аксиоматике теории надежности технических устройств. Однако существуют серьезные отличия между работой ПП и аппаратуры.

Действительно, при эксплуатации технических устройств, даже при неизменных эксплуатационных условиях, всегда существует отличная от нуля вероятность выхода его из строя. В то время, как любое количество запусков программных продуктов при одних и тех же данных будут давать одинаковые результаты. Кроме того, временные затраты на проведение тестовых испытаний часто зависят и от принятой модели тестирования программы и от исходных данных, что может нарушить линейность временного параметра. С другой стороны, эксплуатацию ПП можно, в некотором смысле, считать тестовым испытанием, поскольку при эксплуатации программ обычно выявляется большинство ошибочных ситуаций.

Другое предположение заключается в постулировании зависимости частоты отказов от числа ошибок, оставшихся в ПП, что выглядит вполне логично и подтверждается экспериментальными исследованиями.

В модели надежности Нельсона [52] предлагается вместо временного параметра функции надежности использовать понятие независимых *прогнозов* программы. В этих условиях понятие *надежности ПП* можно интерпретировать как *вероятность безотказного выполнения k независимых прогнозов программы*.

Действительно, теоретически (а иногда и практически) можно представить гипотетический тест, который проверяет работоспособность ПП для всех возможных сочетаний исходных данных, тем более, что у ЭВМ с ограниченной разрядной сеткой таких данных хотя и большое, но конечное число. Проведение такого теста не только точно определяет уровень надежности ПП, но, в случае отсутствия ошибок, позволяет утверждать с абсолютной доверительной вероятностью абсолютную надежность программы, чего невозможно сделать для технических устройств. Из сказанного следует, что уровень надежности ПП зависит не от временного фактора, а от уровня просмотра сочетаний исходных данных ПП, т.е. от количества независимых прогнозов.

3.3.1. Оценка надежности акторов

В предлагаемой модели оценки надежности акторов технологии ГСП сделаны следующие предположения:

1. Акторы, реализующие функциональное отображение вектора исходных данных X на вектор значения функции Y ($Y=f(X)$), являются непрерывными функциями с конечным или счетным числом разрывов первого или второго рода, т.е. принадлежат к классу *вычислительных модулей*.

2. Надежность актора оценивается относительно *арифметических* или *семантических* ошибок. Причем семантические ошибки в модуле идентифицируются по факту "непопадания" вычисляемых параметров модуля в предписанную область значений параметра

3. В БД ПОП заранее описаны область определения актора и область его значений.

Пусть Ω – область определения актора, Ω_E – область ошибок актора (в общем случае многосвязная). Через $q = \frac{V(\Omega_E)}{V(\Omega)}$ определим вероятность возникновения ошибочной ситуации. Здесь $V(.)$ – мера соответствующих множеств (гиперобъем множества).

Надежность актора определим как вероятность успешного выполнения k прогонов актора при независимом для каждого прогона выборе данных из области определения актора:

$$P(k) = (1 - q)^k \quad (3.22)$$

Для вычисления надежности ПП необходимо оценить величину вероятности возникновения ошибочной ситуации q . Для этого воспользуемся схемой независимых испытаний Бернулли, реализованной в алгоритме статистических испытаний актора.

Предположим, что проведена серия из N испытаний, в которой ошибка не была обнаружена, то есть реализовалась серия $\left(\underbrace{0, 0, \dots, 0}_k \right)$. Сделаем допущение, что на $N+1$ прогоне реализовалась ошибка, то есть $\left(\underbrace{0, 0, \dots, 0}_k, 1 \right)$. В этом случае имеем следующую оценку для q :

$$\hat{q} = \frac{1}{N + 1}. \quad (3.23)$$

На самом деле, справедливо неравенство $q < \hat{q}$, поскольку при тестировании предполагалось возникновение ошибочной ситуации на $N+1$ прогоне программы.

Основной проблемой предложенного способа оценивания вероятности q является отсутствие ошибочных ситуаций в процессе реализации независимых испытаний. В схему Бернулли заложена необходимость реализации некоторого количества ошибочных ситуаций. Попытаемся к данной задаче применить механизм Байесовского оценивания q , использующего априорную информацию о статистических свойствах оцениваемого параметра [19].

Пусть по схеме Бернулли произведено N независимых испытаний, в которых ошибка не была обнаружена, т.е. $X = (0, 0, \dots, 0)$, где X – вектор наблюдений. Обозначим через $P(X|q)$ – распределение вектора X при заданном параметре q . Причем $P(X|q) = (1 - q)^N$. Зададим априорную плотность распределения параметра q следующим образом:

$$h_0 = \begin{cases} 1/b & q \in [0, b], \\ 0 & q \notin [0, b] \end{cases} \quad (3.24)$$

Выбрано равномерное распределение оцениваемого параметра, поскольку имеет место полная неопределенность относительно его

предполагаемых значений. С другой стороны, как показывает оценка максимального правдоподобия (3.23), истинное значение параметра q находится на отрезке $[0, \hat{q}]$.

Вычислим нормирующую константу

$$g(X) = \int P(X|q)h_0(q)dq = \int_0^b (1-q)^N \frac{1}{b} dq = \frac{1}{b(N+1)} (1 - (1-b)^{N+1}).$$

Тогда можно вычислить апостериорную плотность распределения:

$$h(q|X) = \frac{P(X|q)}{g(X)} = \frac{(N+1)(1-q)^N}{(1 - (1-b)^{N+1})}. \quad (3.25)$$

Рассмотрим квадратичную функцию потерь, с помощью которой вычислим апостериорный байесовский риск

$$R(d(X), X) = \int_0^b (q - d(X))^2 h(q|X) dq \rightarrow \min_{d(X)}. \quad (3.26)$$

В работе [19] показано, что для выпуклой функции потерь (3.26) такая оценка единственна, причем единственной функцией $d(X)$, минимизирующей $R(d(X), X)$, является апостериорное ожидание параметра q , т.е.

$$\hat{q}_B = \int_0^b q h(q|X) dq = \int_0^b \frac{q(1-q)^N (N+1)}{(1 - (1-b)^{N+1})} dq$$

После взятия интеграла и упрощения формулы имеем:

$$\hat{q}_B = \frac{1}{N+2} - \frac{b(N+1)}{N+2} \left(\frac{(1-b)^{N+1}}{1 - (1-b)^{N+1}} \right). \quad (3.27)$$

Как показали эксперименты (при больших N) оценка (3.27) на порядок меньше оценки максимального правдоподобия, если в качестве b брать оценку максимального правдоподобия (3.23). Например, при $N=1000000$, $\hat{q} = 1.0 \cdot 10^{-6}$, а $\hat{q}_B = 4.18 \cdot 10^{-7}$.

Для оценки параметра q можно использовать график зависимости диаметра области ошибок от размеров разрядной сетки. Например, для зависимости D2 (см. рис. 3.17), при $N=1000000$ испытаний, ошибки должны были бы возникать начиная с $t < 6$, что соответствует диаметру области ошибок $D(\Omega_E) = 1/\sqrt{N} = 0.001$ (для единичного квадрата, в плоскости которого ищется ошибка). Если ошибка не проявилась при $t \in [1; 6]$, то это означает, что реальный диаметр ошибки меньше диаметра области ошибок с минимальным числом операций $D_{\min}(6) = 5 \cdot 10^{-6}$. После приведения размерности разрядной сетки к принятой на ЭВМ, имеем что для базового значения $t=15$ диаметр

области ошибок составляет (в случае их не выявления) $D(\Omega_E) = 5 \cdot 10^{-15}$, откуда $q = V(\Omega_E) / 1 = 2.5 \cdot 10^{-29}$.

Экстраполяция диаметра области ошибок

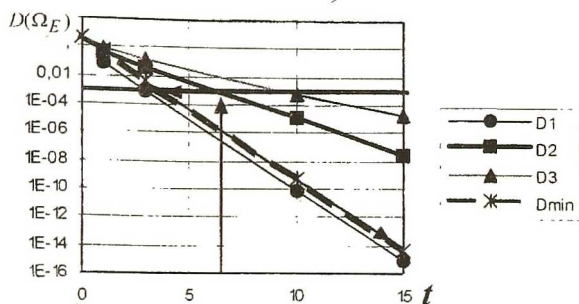


Рис. 3.17

Эффективность тестирования актора становится еще больше (по крайней мере при идентификации ошибок переполнения порядка), если при поиске ошибок воспользоваться алгоритмом адаптивного случайного поиска (см. п. 3.1.2). Принятые допущения относительно свойств актора гарантируют сходимость алгоритма АСП либо к локальному экстремуму, либо к точке разрыва второго рода.

3.3.2. Оценка надежности агрегатов

Агрегат в технологии ГСП представляется графом, в котором объекты связываются некоторой сетевой структурой. Надежность агрегата определяется надежностью составляющих его объектов и корректностью структуры граф-программы [35].

Структурные ошибки при составлении программы чаще всего носят методологический характер и возникают по причине неправильности алгоритма программы или неверного его использования (например, запуск алгоритма оптимизации из области его расходимости). Формальных методов обнаружения такого рода ошибок в настоящее время не существует. Качество тестирования структура программы оценивается на основе анализа маршрутов исполнения программы, например, с помощью оценки полной вероятности правильного функционирования программы [35].

Предположим, что корректность структуры агрегата проверена методами, предложенными в п.3.2., и не вызывает сомнений. Тогда надежность агрегата будет определяться надежностью входящих в него объектов.

Пусть q_i - ненадежность i -го модуля агрегата. Пусть каждый i -й объект агрегата вызывается $\tilde{m}_i$ раз, где

$$\tilde{m}_i = \frac{1}{N} \sum_{k=1}^N m_{ik}, \text{ а } m_{ik} - \text{число вызовов } i\text{-го модуля в } N \text{ тестах.}$$

Ненадежность i -го модуля за $\tilde{m}_i$ прогонов в одном расчете можно оценить величиной

$$Q(\tilde{m}_i) = 1 - (1 - q_i)^{\tilde{m}_i} = Q_i. \quad (3.28)$$

Схему маршрута обозначим $M_j = i_{j_0} i_{j_1} \dots i_{j_k}$. Предположим, что в некотором агрегате G имеется L результирующих маршрутов. Очевидно, что, если на любом из объектов маршрута возникнет ошибка, то весь маршрут можно считать ошибочным. Тогда ненадежность схемы маршрута можно оценить величиной

$$Q_{M_j} = 1 - \prod_{i=1}^{j_k} (1 - Q_{i_i}).$$

Пусть каждая из схем маршрута реализуется при тестировании с вероятностью r_j , причем справедливо $\sum r_j = 1$. Тогда ненадежность агрегата можно оценить величиной

$$Q_G = \sum_{j=1}^L r_j Q_{M_j} \quad (3.29)$$

Упростим формулу (3.28), положив $\tilde{m}_i = m$, $q_i = q$ и предположив, что все маршруты вызываются с одинаковой вероятностью $r_j = 1/L$. Если при этом все маршруты имеют одинаковую длину, то ненадежность агрегата равна

$$Q_G = 1 - (1 - q)^{mn}. \quad (3.30)$$

Следовательно надежность агрегата может быть вычислена по формуле:

$$P_G = (1 - q)^{mn} \quad (3.31)$$

Из формулы (3.31) видно, что надежность агрегата тем выше, чем выше надежность составляющих его модулей, чем проще ее структура (короче программа, меньше в ней маршрутов) чем меньше в ней циклов (меньше значение параметра m).

Пусть $\hat{m} = \frac{1}{L} \sum \tilde{m}_i$ - средняя для всего графа частота вызовов объектов актора; $\hat{n}$ - средняя длина маршрута. Предположим, что агрегат был подвергнут процедуре тестирования методом статистических испытаний и по результатам тестовых испытаний получена оценка вероятности возникновения

ошибки на агрегате q_G . С другой стороны, по характеристикам надежности объектов, составляющих агрегат, подсчитана ненадежность агрегата $Q_G = 1 - (1 - q)^{\hat{m}\hat{n}}$, где q - ненадежность объектов агрегата. Если $Q_G > q_G$, то это означает, что надежность объектов агрегата в области их "совместной работы" под управлением агрегата выше, чем показали автономные испытания. В этом случае надежность объектов может быть перевычислена по формуле:

$$\hat{q} = 1 - \hat{m}\hat{n}\sqrt{1 - q_G}. \quad (3.32)$$

Уточненные оценки надежности объектов агрегата справедливы на области данных совместной работы объектов, а она может оказаться значительно меньше, чем на области определения каждого из объектов. Например, тестирование актора down, входящего в состав алгоритма адаптивного случайного поиска, производилась по параметрам Sibaz-Ntek в диапазоне $[0; 10^{18}] \times [0; 10000]$, в то время как область совместной работы по результатам тестовых испытаний над функцией Розенброка представляет собой прямоугольник $[0; 60] \times [0; 300]$.

Более точные значения поправок значений оценок ненадежностей объектов агрегата, с учетом их совместной работы, можно получить, если решить следующую условную задачу оптимизации:

$$\begin{cases} \sum (q_i - \hat{q}_i)^2 \rightarrow \min_{\hat{q}_i} \\ Q(q_1, q_2, \dots, q_m) = Q_G \end{cases} \quad (3.33)$$

Критерий оптимизации (3.33) позволяет найти ближайшие к $\hat{q}_i$ оценки ненадежности объектов агрегата при условии полученной оценки ненадежности агрегата Q_G .

ИСПОЛЬЗОВАНИЕ ТЕХНОЛОГИИ ГСП ДЛЯ ПОСТРОЕНИЯ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ САПР

В этой главе приводятся примеры построения подсистем программного обеспечения САПР ГТД, реализованные с использованием методов и инструментальных средств технологии ГСП.

В плане развития методов принятия решений, используемых на начальных этапах проектирования авиационного двигателя, приводятся оригинальные методы решения задачи многокритериальной оптимизации в условиях неопределенности исходной проектной информации.

4.1 Примеры использования технологии ГСП для автоматизации начальных этапов проектирования авиационных двигателей

Сфера обработки информации становится узким местом в общественном производстве, во многом определяя перспективы его развития [10]. В сфере проектирования новые информационные технологии связаны с автоматизацией научных исследований. Системам автоматизации проектных работ свойственны проблемы разного уровня сложности и требующие скорейшего своего решения.

Так при автоматизации процесса проектирования авиационного газотурбинного двигателя необходимо решить целый комплекс взаимосвязанных проблем, основными из которых являются следующие [41]:

1. Обоснование рациональной структуризации процесса проектирования, которая зависит от сложности рассматриваемого типа ГТД и степени детализации проектно-конструкторских задач.
2. Формализация описания процессов проектирования двигателя, его математических моделей.
3. Организация выбора проектного решения в условиях многокритериальности, неопределенности исходных данных и наличия неформализуемых факторов ЛПР. Обоснование принимаемых решений.

Системный подход предполагает изучение объекта проектирования, например системы ГТД-ЛА как единого целого. В тоже время при системном подходе обычно реализуется декомпозиция сложной системы на относительно самостоятельные подсистемы. Каждая из этих подсистем может быть расчленена на подсистемы более низкого уровня и т.д. Последовательное решение иерархических задач позволяет принимать проектные решения, обеспечивая по мере перехода на более низкие иерархические уровни от ЛА к ГТД и его элементам все большую степень детализации проектируемого объекта [41]. Математические модели и процедуры принятия решений

необходимые для выполнения каждой проектной операции в процессе проектирования могут существенно меняться как по уровню сложности, так и степени детализации. Поэтому предварительная заготовка таких моделей под каждую конкретную задачу невозможна. В этих условиях необходимо обеспечить гибкое и оперативное формирование требуемых расчетных моделей из имеющихся в программном обеспечении блоков. Требуется некоторое *средство автоматизации процессов построения и модификации* программного обеспечения САПР. В качестве такового предлагается использовать технологию ГСП.

В разделе на примере построения процедур принятия проектных решений на начальном этапе проектирования ГТД продемонстрированы возможности предлагаемой технологии ГСП.

4.1.1. Многокритериальная задача оптимизации при выборе параметров сложных технических систем

Существующие процедуры принятия решений рассматривают *решение* как сознательный выбор одной альтернативы из множества альтернатив. Этот выбор осуществляет лицо, принимающее решение (ЛПР), которое обладает правами принятия решения и в тоже время несет ответственность за его последствия.

В случае наличия единственного критерия оценки качества решения выбор оптимального решения реализуется в результате параметрической оптимизации целевой функции (критерия эффективности). В многокритериальной задаче оптимизации сравнение решений производится не по одному критерию, а с учетом нескольких факторов - числовых функций $f_1, f_2, \dots, f_n$, оценивающих качество проектируемой системы с различных позиций. В общем случае компонентами моделей $f_i(X, B)$ являются оптимизируемые параметры $X = (x_1, x_2, \dots, x_m)$ и случайные и неопределенные факторы (исходные данные) $B = (b_1, b_2, \dots, b_r)$. В том случае, когда вектор B задан, имеет место задача принятия решений *в условиях определенности*.

Рассмотрим многокритериальную задачу принятия решений в условиях определенности (проблемы решения аналогичной задачи в условиях неопределенности обсуждаются в п.4.1.3). Будем считать, что для каждого критерия $f_i(X)$ определено множество I_i , из которого он принимает свои значения. Частные критерии f_i образуют векторный критерий $f = (f_1, f_2, \dots, f_n)$. Считается, что каждое решение X полностью характеризуется соответствующей векторной оценкой. Выбор оптимального (точнее рационального) решения сводится к выбору оптимальной оценки из множества достижимых оценок

$$Y = f(X) = \{y \in E^n | y = f(X), X \in \Omega_X\},$$

где E^n - n -мерное числовое пространство, Ω_X - множество всех альтернатив.

Относительно критериев эффективности f_i положим, что они являются *выпуклыми, унимодальными* функциями, достигающими своих минимальных значений $f_{i\min}$. Кроме целевых функций f_i при выборе оптимальных решений часто необходимо учитывать ограничения $g(X) = (g_1(X), g_2(X), \dots, g_p(X)) \leq 0$. Таким образом, многокритериальную задачу оптимизации параметров сложной технической системы в условиях определенности можно сформулировать так

$$\begin{cases} f(X) \xrightarrow{X} \text{opt}, \\ g(X) \leq 0. \end{cases} \quad (4.1)$$

В многокритериальных задачах оптимизации оптимальное решение *не единственно*, что порождает большое количество различных методов решения задачи, отличающихся и идейным содержанием и техническим исполнением, но которые невозможно сравнить между собой по качеству принимаемых решений. В тоже время существует единственный объективный принцип сужения множества достижимых оценок до, так называемых, *эффективных* или *Парето-оптимальных* оценок [46].

Принцип основан на введении в множестве достижимых оценок отношения *частичного порядка* $\leq$, естественным образом определяющего отношение предпочтения. В многокритериальной задаче оптимизации из двух векторных оценок, отличающихся лишь одной компонентой, предпочтительнее та, у которой такая компонента меньше.

На множестве достижимых оценок Y зададим отношение порядка ($\leq$) следующим образом:

$$y^1 \leq y^2, \text{ если } y_i^1 \leq y_i^2, \quad i = 1, 2, \dots, n \text{ и } y^1 \neq y^2.$$

Минимальную по ($\leq$) относительно Y оценку $\bar{y} \in Y$ называют *эффективной (Парето-оптимальной) оценкой*.

Множество всех эффективных оценок $\bar{y} \in Y$ называется *Парето-оптимальным множеством* P_Y .

Обычно множество эффективных решений оказывается достаточно богатым и содержит существенно разные решения, поэтому для выбора окончательного решения необходимо привлекать дополнительную информацию (в последнее время такой выбор реализуется с привлечением ЛПР). В тоже время множество эффективных оценок значительно уже чем исходное множество достижимых оценок и содержит объективно лучшие оценки. В этой связи понятие эффективной оценки играет важнейшую роль в теории многокритериальной оптимизации [46]. Действительно, если и выбирать оптимальное решение, то кроме иных положительных качеств (с точки зрения ЛПР) оно должно как минимум обладать свойством эффективности. Поэтому построение множества эффективных оценок является одним из первых этапов большого числа процедур и методов многокритериальной оптимизации.



Рис. 4.1 Классификационная схема методов многокритериальной оптимизации

Методы анализа многокритериальных задач многочисленны и разнообразны. На рисунке 4.1 представлена классификационная схема методов многокритериальной оптимизации (схема заимствована из работы [36]). Там же приводится подробный анализ представленных методов многокритериальной оптимизации.

В настоящее время можно выделить два основных направления развития теории принятия решений: аксиоматический подход к решению многокритериальных задач оптимизации и прагматические методы выбора рациональных решений. Глобальной целью первого подхода является полная автоматизация процедуры принятия решения за счет аксиоматического описания предпочтений ЛПР и исключения последнего из процедуры выбора решений. Достаточно полно это направление представлено в работах [44, 49]. Аксиоматические методы чрезвычайно сложны и вызывают значительные затруднения при их использовании. Кроме того, вызывает серьезные сомнения возможность адекватного аксиоматического описания предпочтений ЛПР,

поскольку последний (с системотехнических позиций) является более противоречивой и более нелинейной системой чем проектируемая техническая система. И, наконец, ЛПР никогда не переложит ответственность за принятие решения на формально-математические методы, как бы красиво они не выглядели бы.

Второе направление исходит из позиции создания удобного инструмента для проведения исследований по выбору решений с участием ЛПР. Методы этой группы основаны на активном диалоге ЛПР с вычислительной машиной. Диалоговые процедуры принятия решения являются наиболее перспективным направлением и представляют собой итеративный процесс обмена информацией между человеком и ЭВМ.

Список методов многокритериальной оптимизации обширен и многообразен. Невозможно перечислить всех авторов, внесших посильный вклад в развитие этой науки. Не желая никого обидеть, все же приведем несколько наиболее известных в этой области авторов. Начало исследований в этой области положили работы Фон Неймана Дж. и Моргенштерна О., значительный вклад в теорию принятия решений внесли такие ученые как Х. Райфа, Льюс А., Джоффрион А., Полак Е. И т.д. В нашей стране ранее проблеме выбора решений уделялось значительное внимание, широко известны работы Гермейера Ю.Б., Моисеева Н.Н., Статникова Р.Б., Соболя И.М., Краснощекова П.С., Подиновского В.В., Ногина В.Д., Ларичева О.И., Лотова А.В. и многих других авторов.

Большинство методов выделения множества эффективных решений основано на принципе *скаляризации* исходной задачи, т.е. замене векторного критерия соответствующей скалярной параметрической задачей математического программирования. Например, задача (4.1) в соответствии с теоремой Гермейера [11] совпадает с множеством решений параметрической задачи минимизации функции (если не учитывать ограничения)

$$\max_i \mu_i f_i(X),$$

где вектор μ принадлежит некоторому множеству M . В некоторых случаях используется «свертка» критериев вида $\left[\sum_i (\mu_i f_i)^s \right]^{1/s}$. Для выпуклых многокритериальных задач в [46] показано, что множество эффективных оценок может быть построено в результате минимизации функции $\sum_i \mu_i f_i$ для различных значений вектора μ . Однако представленные подходы лишь косвенно описывают множество эффективных оценок. Не всегда тривиальны взаимосвязи «весовых» векторов μ с Парето-оптимальными оценками и при построении множества эффективных оценок возникают многочисленные технические трудности. В работах [22, 32] предлагается идея прямого метода описания Парето-оптимального множества оценок. Идея метода основывается на использовании «разделяющей» функции [22].

Введем нормализованную векторную функцию

$$y(X) = \{y_1(X), y_2(X), \dots, y_{n+p}(X)\}, \quad (4.2)$$

где

$$y_i(X) = (f_i(X) - \min_X f_i(X)) / \min_X f_i(X), \quad i = 1, 2, \dots, n \quad \text{для критериев}$$

эффективности и

$$y_i(X) = (g_i(X) - g_{i \min}) / g_{i \min}, \quad i = n+1, \dots, n+p \quad \text{для ограничений.}$$

Здесь $g_{i \min}$ - предельный уровень функции ограничения, ниже значения которого ограничение не задается.

Для ограничения типа «равенства» нормированный показатель можно задать следующим образом:

$$y_i(X) = (g_i(X) - g_i^*)^2 + 1,$$

где g_i^* - уровень ограничения.

Все компоненты векторной целевой функции $y(X)$, независимо от того относятся ли они к целевым функциям или являются ограничениями, достигают своего наименьшего значения в точке 0 и в этом смысле семантически неразличимы. Далее эффективное множество оценок будем строить для векторной целевой функции $y(X)$ какие бы компоненты она не содержала бы. При таком подходе реализуется решение задачи (4.1) в самых разнообразных постановках (безусловной или условной оптимизации).

Определим «разделяющую» функцию следующим образом:

$$G(\Delta y) = \min_X \max_i \frac{y_i(X)}{\Delta y_i} - 1, \quad (4.3)$$

где Δy_i определяет величину относительного нормализованного критериального отклонения от своего наименьшего значения.

Функция $G(\Delta y)$ построена таким образом, что уравнение

$$G(\Delta y) = 0 \quad (4.4)$$

описывает множество эффективных оценок.

Построенная функция $G(\Delta y)$, описывая множество эффективных оценок с помощью нелинейного уравнения (4.4), предоставляет возможность применения готовых традиционных графических пакетов для изображения Парето-оптимального множества. Докажем, что уравнение (4.4) описывает Парето-оптимальное множество оценок.

Утверждение 4.1 Оценка $\Delta y^* = (\Delta y_1^*, \Delta y_2^*, \dots, \Delta y_n^*)$ эффективна тогда и только тогда, когда минимальное по критерию

$$\max_i (y_i(X) / \Delta y_i^*) \quad (4.5)$$

решение X^* удовлетворяет условиям

$$\frac{y_1(X^*)}{\Delta y_1^*} = \frac{y_2(X^*)}{\Delta y_2^*} = \dots = \frac{y_n(X^*)}{\Delta y_n^*} = 1, \quad (4.6)$$

где $X^* \in \Omega_X$, а $y_i(X)$ - непрерывные, выпуклые, унимодальные функции.

Доказательство:

Необходимость. Пусть Δy° - эффективная оценка, то есть не существуют другие оценки такие, что $\Delta y \leq \Delta y^\circ$. Тогда для оптимального по (4.5) решения X^* будет выполняться условие (5.6). Действительно, пусть для некоторого j оно не

выполняется, то есть $\frac{y_j(X^*)}{\Delta y_j^\circ} < 1$ или $y_j(X^*) < \Delta y_j^\circ$, но тогда оценка

$y(X^*) \leq \Delta y^\circ$, что противоречит условию эффективности оценки Δy° . Случай

$\frac{y_j(X^*)}{\Delta y_j^\circ} > 1$ противоречит условию формирования минимакса в точке X^* (4.5).

Достаточность. Пусть для оптимального решения X^* выполняется условие (5.6). Предположим, что существует $\Delta y^1 \leq \Delta y^\circ$. В силу непрерывности функций $y_i(X)$ для оценки Δy^1 минимум критерия (5.5) достигается для другого решения X^{**} ($X^{**} \neq X^*$).

Пусть для любых i справедливо $y_i(X^{**}) < y_i(X^*)$, тогда

$$\max_i \frac{y_i(X^{**})}{\Delta y_i^\circ} < \max_i \frac{y_i(X^*)}{\Delta y_i^\circ}, \quad (4.7)$$

что противоречит условию формирования минимакса по критерию (4.5), а следовательно неверно, что $\Delta y^1 \leq \Delta y^\circ$.

Предположим теперь, что существует j , для которого $y_j^1(X^{**}) = y_j(X^*)$, но тогда X^* и X^{**} два равноценных решения задачи минимизации функции

$\max_i \frac{y_i(X)}{\Delta y_i^\circ}$, которые в точках X^* и X^{**} совпадают с $\frac{y_i(X^*)}{\Delta y_i^\circ}$. В тоже время

выпуклая унимодальная функция (как композиция выпуклых унимодальных функций $y_i(X)$) может иметь только одно оптимальное решение, следовательно. $X^{**} = X^*$ или $\Delta y^1 = \Delta y^\circ$.

Утверждение 4.2. Для выпуклых непрерывных унимодальных функций (4.2) справедливо включение

$$\Pi_Y \subset \{\Delta y \mid G(\Delta y) = 0\},$$

где $G(\Delta y)$ - «разделяющая» функция (4.3).

Доказательство:

Пусть Δy° - эффективная оценка, тогда на основании теоремы 5.2 для нее справедливо

$$\frac{y_1(X^*)}{\Delta y_1^0} = \frac{y_2(X^*)}{\Delta y_2^0} = \dots = \frac{y_n(X^*)}{\Delta y_n^0} = 1, \text{ то есть } \min_X \max_i \frac{y_i(X)}{\Delta y_i^0} = \frac{y_j(X^*)}{\Delta y_j^0} = 1. \text{ Тогда}$$

из определения «разделяющей» функции имеем $G(\Delta y) = 0$, из чего следует справедливость включения $\Pi_Y \subset \{\Delta y | G(\Delta y) = 0\}$.

Утверждение 4.1 имеет и самостоятельное значение, поскольку в отдельных случаях позволяет выводить аналитические выражения для описания эффективного множества оценок.

Действительно, из утверждения 4.1 следует, что задачу вычисления эффективной оценки можно свести к решению системы нелинейных уравнений

$$\begin{cases} y_1(X) = \Delta y_1 \\ y_2(X) = \Delta y_2 \\ \dots \\ y_n(X) = \Delta y_n \end{cases} \quad (4.8)$$

где Δy - вектор ожидаемых отклонений критериев эффективности от своих оптимальных значений.

Если размерность оптимизируемых параметров меньше или равно размерности вектора оценок ($n \leq m$), то имеет место недоопределенная система нелинейных уравнений (система имеет множество решений). Доопределить систему можно, если поставить задачу условной оптимизации

$$\min_X (y_1(X) / \Delta y_1)$$

при ограничениях (4.8).

В тех случаях, когда $n > m$, система имеет единственное решение X^* и задача построения эффективной оценки сводится к задаче решения системы уравнений (4.8) с последующим вычислением эффективной оценки по формулам $\Delta y_1^0 = y_1(X^*)$, $\Delta y_2^0 = y_2(X^*)$, ..., $\Delta y_n^0 = y_n(X^*)$.

Рассмотрим следующий иллюстративный пример.

Пример 1

Пусть в нормализованной форме заданы критерий эффективности

$$y_1(x, y) = (x - 1)^2 + y^2, \quad (4.9)$$

$$y_2(x, y) = x^2 + y^2.$$

Из условия (4.5) имеем

$$\begin{cases} y_1(x, y) = (x - 1)^2 + y^2 = \Delta y_1 \\ y_2(x, y) = x^2 + y^2 = \Delta y_2 \end{cases}, \quad (4.10)$$

Вычитая второе уравнение из первого имеем

$$x^* = \frac{1 + \Delta y_2 - \Delta y_1}{2}.$$

Зафиксировав Δy_1 из второго уравнения (4.10) и условия формирования эффективной оценки (4.5) имеем $\frac{(x^*)^2 + y^2}{\Delta y_2} \rightarrow \min_y$, откуда $y^* = 0$.

Подставив полученный результат во второе уравнение (4.10)? получаем уравнение множества эффективных оценок

$$(\Delta y_1^\circ - \Delta y_2^\circ)^2 - 2(\Delta y_1^\circ + \Delta y_2^\circ) + 1 = 0. \quad (4.11)$$

Пример 2

Систему критериев (4.9) расширим еще одним критерием

$$y_3(x, y) = x^2 - (y - 1)^2,$$

в результате получим систему уравнений

$$\begin{cases} y_1(x, y) = (x - 1)^2 + y^2 = \Delta y_1 \\ y_2(x, y) = x^2 + y^2 = \Delta y_2 \\ y_3(x, y) = x^2 + (y - 1)^2 = \Delta y_3 \end{cases} \quad (4.12)$$

Система (4.12) переопределена, поэтому «жестко» зададим любые два критериальных отклонения, например, $\Delta y_1^\circ, \Delta y_2^\circ$ (последнее критериальное отклонение определяется из условия формирования минимакса). Кроме того, положим $\Delta y_1^\circ = \Delta y_2^\circ = \Delta y^\circ$.

Решив систему

$$\begin{cases} y_1(x, y) = (x - 1)^2 + y^2 = \Delta y^\circ \\ y_2(x, y) = x^2 + y^2 = \Delta y^\circ \end{cases}$$

имеем

$$\begin{cases} x^* = 1/2, \\ y^* = \pm \sqrt{4\Delta y^\circ - 1}, \end{cases}$$

Подставив в последнее уравнение системы (4.12) получим уравнение

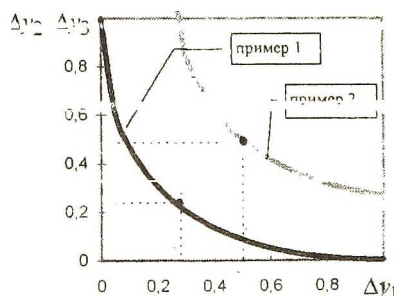


Рис.4.2 Множества эффективных оценок примеров 1 и 2.

множества эффективных оценок

$$(\Delta y_3^* - \Delta y^* - 1)^2 - 4(\Delta y^* - \frac{1}{4}) = 0.$$

На рисунке 4.2 представлены множества эффективных оценок, построенные для примеров 1 и 2. На рисунке 4.3 для примера 2 показано формирование эффективного решения для критериальных отклонений $\Delta y_1^* = \Delta y_2^* = \Delta y_3^* = 0.5$.

4.1.2 Аппроксимация множества эффективных оценок

Начальный этап проектирования авиационного двигателя обычно связан с его термогазодинамическим расчетом, на котором закладываются расчетные параметры рабочего процесса двигателя, обеспечивающие наиболее выгодное использование свободной энергии ГТД [41]. При этом качество ГТД оценивается, исходя из системных критериев, учитывающих интересы как двигателя, так и самолета, на котором он установлен. Так

например, экономическое совершенство дозвукового магистрального самолета оценивается по комплексу таких критериев как: затраты топлива на 1 т.км (Сткм), взлетная масса (M_0), стоимость жизненного цикла (СЖЛ) и т.д. Кроме того, могут использоваться и внутренние критерии, оценивающие

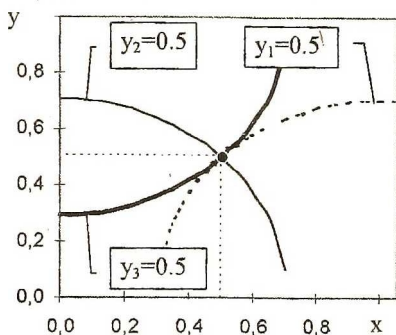


Рис. 4.3 Формирование эффективного решения

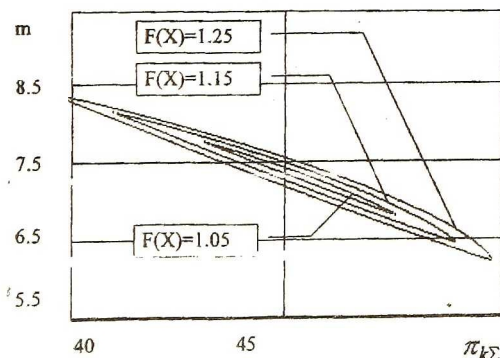


Рис. 4.4 Изолинии функции $F(X)$ для формирования минимакса критериев M_0 и Sp

эффективность силовой установки, например, удельный расход топлива (Суд). Исследование свойств критериев оценки эффективности системы ГТД-ЛА показывают, что они являются выпуклыми унимодальными функциями с пологим «дном», изолинии которых имеют вытянутую эллипсоидальную форму [41]. Другими словами целевые функции имеют ярко выраженный «овражный» характер.

Наличие оврагов в значительной степени осложняет реализацию процедур оптимизации самих критериев эффективности, что же касается минимизации «свертки» критериев $F(X) = \max_i \frac{y_i(X)}{\Delta y_i} - 1$, используемой при построении «разделяющей» функции, то эффект «овражности» только усиливается. «Стенки» оврага становятся более крутыми (по сравнению с целевыми функциями), дно - более пологим (см. рис.4.4). Более того, функция $F(X)$ по линии дна оврага не является дифференцируемой функцией, что ограничивает возможности при использовании методов оптимизации, например, неудовлетворительные результаты дают градиентные методы оптимизации.

Наиболее подходящим методом оптимизации оказался метод деформированных многогранников Нелдера и Мида [24], который легко адаптируется к особенностям оптимизируемой функции, не «замечает» отдельных «шероховатостей» функции, вызванных погрешностями вычислений, не критичен к гладкости оптимизируемой функции. Однако при минимизации функции $F(X)$, в силу ее овражности, метод часто «сворачивает» многогранник задолго до достижения минимума функции, при этом в среднем реализуется 100...200 обращений к функции. Применение специальных мероприятий для устранения эффекта овражности увеличивает среднее число обращений к функции до 2500...3000. Если учесть, что каждое обращение к «свертке» целевых функций обеспечивается полным термогазодинамическим расчетом двигателя с использованием сложной математической модели ГТД и требует значительных затрат машинного времени, а для нахождения одного из решений уравнения $G(\Delta y) = 0$, итерационно решается несколько (10..20) задач минимизации функции $F(X)$, то общая сложность решения задачи нахождения эффективной оценки критериев эффективности составляет значительную величину. Естественным выходом из сложившегося положения дел является построение приближенной модели «разделяющей» функции, то есть аппроксимация Парето-оптимального множества эффективных оценок..

В окрестностях эффективной оценки $\Delta y^0 = (\Delta y_1^0, \Delta y_2^0, \dots, \Delta y_n^0) \in P_f$ разложим функцию $G(\Delta y)$ в ряд Тейлора:

$$G(\Delta y) \approx \frac{1}{2} (H(\Delta y^0) \Delta, \Delta) + (grad(\Delta y^0), \Delta), \quad (4.13)$$

где $\Delta = \Delta y - \Delta y^0$;

$H(\Delta y^0)$ - гессиан функции $G(\Delta y)$ в точке Δy^0 ;

$\text{grad}(G(\Delta y^0))$ - градиент функции $G(\Delta y)$ в точке Δy^0 .

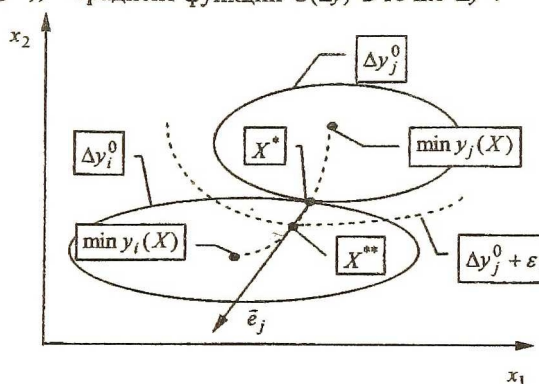


Рис. 4.5 Формирование направления перемещения минимаксного решения X^* при вариации критериального отклонения Δy_j^0

Принципиально возможно вычисление частных производных функции $G(\Delta y)$ с помощью конечно-разностных формул. Однако овражный характер функции $F(X)$ и связанная с этим неустойчивость результата решения минимаксной задачи порождают значительные погрешности при вычислении частных производных. Рассмотрим альтернативный способ решения поставленной задачи.

Пусть Δy^0 - точка разложения «разделяющей» функции (эффективная оценка), а X^* - соответствующее ей эффективное решение (см. рис.4.5). Если критериальному отклонению Δy_j^0 дать приращение ε , то точка формирования минимакса X^* переместится в положение X^{**} , определив направление изменения положения минимакса:

$$\bar{e}_j = \lim_{\varepsilon \rightarrow 0} (X^* - X^{**}) / |X^* - X^{**}|. \quad (4.14)$$

Частную производную «разделяющей» функции можно вычислить по формуле

$$\begin{aligned} \frac{\partial G(\Delta y)}{\partial \Delta y_j} &= \lim_{\varepsilon \rightarrow 0} \frac{G(\Delta y_1, \dots, \Delta y_j + \varepsilon, \dots, \Delta y_n) - G(\Delta y)}{\varepsilon} = \\ &= \lim_{\varepsilon \rightarrow 0} \left(\frac{y_i(X^{**})}{\Delta y_i} - \frac{y_i(X^*)}{\Delta y_i} \right) / \varepsilon, \end{aligned} \quad (4.15)$$

поскольку в точке формирования минимакса в соответствии с утверждением

5.2 выполняется условие $G(\Delta y) = \frac{y_i(X^*)}{\Delta y_i} - 1$.

При $\varepsilon \rightarrow 0$ из утверждения 5.2 в точке минимакса X^{**} имеем

$$\frac{y_j(X^{**})}{\Delta y_j + \varepsilon} = \frac{y_i(X^{**})}{\Delta y_i} \quad \text{или}$$

$$\varepsilon = \Delta y_i \frac{y_j(X^{**})}{y_i(X^{**})} - \Delta y_j. \quad (4.16)$$

В направлении вектора $\bar{e}_j$ по формуле Тейлора разложим функцию $\frac{y_j(X)}{y_i(X)}$:

$$\frac{y_j(X^{**})}{y_i(X^{**})} = \frac{y_j(X^*)}{y_i(X^*)} + \left(\frac{y_j(X^*)}{y_i(X^*)} \right)'_{\bar{e}_j} h + O(h^2), \text{ тогда с учетом (4.16) имеем}$$

$$\varepsilon \approx \Delta y_i \left(\frac{y_j(X^{**})}{y_i(X^{**})} \right)'_{\bar{e}_j} h$$

Введем функцию $q_{j\bar{e}_i} = \Delta y_i \left(\frac{y_j(X^*)}{y_i(X^*)} \right)'_{\bar{e}_j}$ ($q_j(X^*) = \Delta y_i \frac{y_j(X^*)}{y_i(X^*)}$),

тогда для (4.15) имеем

$$\frac{\partial G(\Delta y)}{\partial \Delta y_j} = \frac{1}{\Delta y_i} \lim_{\varepsilon \rightarrow 0} \frac{\left[y_i(X^*) \right]'_{\bar{e}_j} h}{q_{j\bar{e}_i} h} = \frac{y_i' \bar{e}_j(X^*)}{\Delta y_i q_{j\bar{e}_i}(X^*)}.$$

Таким образом, частные производные «разделяющей» функции приближенно могут быть вычислены по формуле:

$$\frac{\partial G(\Delta y)}{\partial \Delta y_j} = \frac{y_i' \bar{e}_j(X^*)}{\Delta y_i q_{j\bar{e}_i}(X^*)} \quad (4.17)$$

В свою очередь, производные по направлению вектора $\bar{e}_j = (e_j^1, \dots, e_j^m)$ вычисляются по следующим формулам:

$$y_i' \bar{e}_j(X) = \sum_{k=1}^m \frac{\partial y_i(X)}{\partial x_k} e_j^k, \quad (4.18)$$

$$q'_{j\bar{e}_j}(X) = \sum_{k=1}^m \frac{\partial q_j(X)}{\partial x_k} e_j^k.$$

Нетрудно видеть, что при вычислении частных производных «разделяющей» функции по формулам (4.17-4.18) используются достаточно гладкие функции $y_1(X), \dots, y_n(X)$, что уменьшает погрешность вычислений. Кроме того, вычислительные эксперименты с математическими моделями ГТД показали, что формулы (4.14) для направляющих векторов $\bar{e}_j$ можно заменить на приближенные формулы:

$$\bar{e}_j \approx (X^* - X_{j\min}) / |X^* - X_{j\min}|,$$

где $X_{j\min}$ - точка минимума j -го критерия эффективности.

Аналогично можно получить приближенные формулы для вычисления вторых частных производных (применяя теорему 4.1 и используя формулу Тейлора):

$$\begin{aligned} \frac{\partial^2 G(X)}{\partial \Delta y_j^2} &= \lim_{\varepsilon \rightarrow 0} \frac{\frac{y_j(X^{***})}{\Delta y_j + \varepsilon} - 2 \frac{y_j(X^*)}{\Delta y_j} + \frac{y_j(X^{**})}{\Delta y_j - \varepsilon}}{\varepsilon^2} = \\ &= \frac{1}{\Delta y_j} \lim_{\varepsilon \rightarrow 0} \frac{y_j(X^{***}) - 2y_j(X^*) + y_j(X^{**})}{\varepsilon^2} = \\ &= \frac{1}{\Delta y_j} \lim_{\varepsilon \rightarrow 0} \frac{y'_{i\bar{e}_j}(X^*)h - y'_{i\bar{e}_j}(X^{**})h}{(q'_{j\bar{e}_j})^2 h^2} = \frac{1}{\Delta y_j} \frac{y''_{i\bar{e}_j\bar{e}_j}(X^*)}{(q'_{j\bar{e}_j})^2}. \end{aligned}$$

Таким образом, имеем

$$\frac{\partial^2 G(X)}{\partial \Delta y_j^2} = \frac{1}{\Delta y_j} \frac{y''_{i\bar{e}_j\bar{e}_j}}{(q'_{j\bar{e}_j})^2}. \quad (4.19)$$

Использование утверждения 4.2 для вычисления смешанных вторых частных производных не представляется возможным. Однако доопределить компоненты матрицы Гессе можно, используя свойства «разделяющей» функции. Например, если функция $G(X)$ непрерывна вместе со своими частными производными первого и второго порядка, то справедливо равенство

$$\frac{\partial^2 G(X)}{\partial \Delta y_i \partial \Delta y_j} = \frac{\partial^2 G(X)}{\partial \Delta y_j \partial \Delta y_i}$$

С другой стороны из утверждения 4.2 следует, что функция $G(X)$ всегда меньше нуля и не может принимать положительные значения, кроме того, «разделяющая» функция не имеет минимума и убывает до бесконечности. Последнее возможно, если матрица квадратичной формы, с помощью которой аппроксимируется $G(X)$, вырождена. Для «разделяющей» функции можно

показать, что смешанные частные производные принимают отрицательные значения. Кроме того, в качестве дополнительной информации можно использовать значения «разделяющей» функции, вычисленные в точках экстремумов целевых функций: $G(0, \Delta y_2^1, \dots, \Delta y_n^1)$, $G(\Delta y_1^2, 0, \dots, \Delta y_n^2), \dots, G(\Delta y_1^n, \dots, 0)$.

В качестве примера рассмотрим задачу построения Парето-оптимального множества оценок перспективных ТРДД для дозвукового магистрального самолета ($H=11$ км, $V=850$ км.ч, $T_{r,kr}^*=1400$ К, $L_p=9000$ км). В качестве оптимизируемых параметров использовались степень двухконтурности (m), степень повышения давления вентилятора (π_b^*) и суммарная степень повышения давления компрессора (π_{Σ}^*). Эффективность системы ГТД-ЛА оценивалась по системным критериям M_0 и $S_{уд}$.

На рисунке 4.6 представлены результаты исследования эффективности дозвукового магистрального самолета. Множество эффективных оценок построено в результате решения нелинейного уравнения $G(X)=0$ с использованием математической модели системы ГТД-ЛА [41]. Там же приводятся результаты аппроксимации Парето-оптимального множества оценок квадратичной формой в точке разложения $\Delta C_{уд} = \Delta M_0 = 0.0138$. По формуле (5.17) был вычислен градиент функции $G(X)$: $(grad(G(\Delta y^0))) = (-34.43; -37.56)$. При вычислении Гессиана «разделяющей» функции

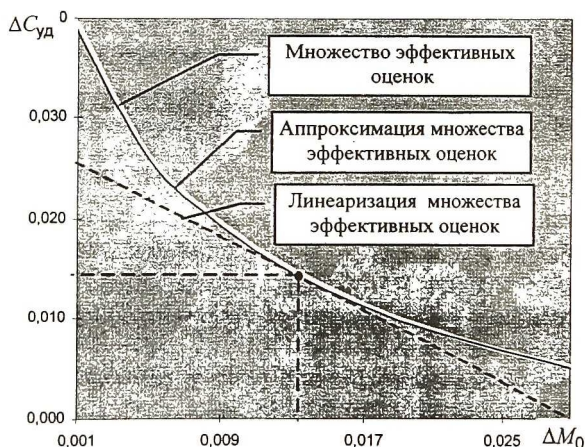


Рис.4.6 Аппроксимация Парето-оптимального множества эффективных оценок ТРДД для ДМС

использовались формула (4.19) и условие вырожденности матрицы Гессиана $\det H(\Delta y) = 0$:

$$H(\Lambda y) = \begin{bmatrix} 623.3 & -668 \\ -668 & 720.8 \end{bmatrix}.$$

Как видно из рисунка квадратичная форма (4.13) достаточно точно воспроизводит множество эффективных оценок. Для нахождения точки разложения «разделяющей» функции требуется приблизительно 100..200 обращений к модели двигателя (решение одной минимаксной задачи типа $\min F(X)$), а для построения аппроксимативной функции типа (4.13) еще 2" обращений к модели двигателя, что в среднем в 300 раз меньше (по трудоемкости), чем при построении двумерной модель множества эффективных оценок с использованием уравнения $G(X)=0$. Кроме того, в результате аппроксимации получается аналитическая модель Парето-оптимального множества оценок.

4.1.3. Неопределенность в задачах выбора параметров рабочего процесса авиационных ГТД

Проектирование сложных технических систем (в том числе и ГТД) обычно происходит в условиях неопределенности исходной проектной информации. Недостаточная определенность значений многих исходных проектных данных обычно связана с существующей в КБ практикой «закладывания» (для перспективного изделия) более высоких, чем достигнутые на настоящий момент времени, уровней совершенства узлов двигателя. Проектирование двигателя без ориентации на будущие технологические достижения делает эту работу экономически невыгодной, поскольку через 4-6 лет созданный новый двигатель не будет отвечать достигнутому к тому времени уровню технического совершенства. С другой стороны невозможно точно предсказать уровень технического и технологического развития общества на достаточно дальнюю перспективу в 4-8 лет.

Проблема принятия решений в условиях неопределенности по своей сложности значительно превосходит эту задачу в детерминированном случае. По способу учета неопределенности исходной проектной информации можно выделить две группы методов.

Наиболее исследован случай стохастически определенной исходной информации [7, 47, 50], когда известны законы распределения исходных данных.

Более сложным представляется случай отсутствия информации о законах распределения исходных данных (полная неопределенность). В этом случае, либо задаются субъективными законами распределения [49], либо исходят из теоретико-игровых способов раскрытия неопределенности[7], применяя критерии типа Вальда, Сэвиджа, Гурвица и т.д.

Пусть $f_i(X, B)$ - критерии оценки эффективности авиационного двигателя, где X оптимизируемые параметры, B неопределенные исходные данные. Предположим, что относительно вектора B известно: его область определения Ω_B и совместный закон распределения данных $\varphi_B(B)$. Следуя рекомендациям Е.С. Вентцель [7], неопределенность в исходной информации можно раскрыть, введя интегральные оценки типа

$$y_i(X) = \int_{\Omega_B} f_i(X, B) \varphi_B(B) \mu_B, \quad (4.20)$$

оценивающие эффективность двигателя по i -му показателю *в среднем*. С помощью такого приема задача о выборе решения в условиях неопределенности превращается в задачу выбора решений в условиях определенности.

Похожий результат получается, если предварительно определиться со средним уровнем совершенства узлов двигателя (средним уровнем значений исходных данных):

$$\bar{B} = \int_{\Omega_B} \varphi_B(B) d\mu, \quad (4.21)$$

а «свертку» неопределенности производить по формуле

$$y_i(X) = y_i(X, \bar{B}). \quad (4.22)$$

Причем $\bar{B}$ может быть и прогнозируемым уровнем оценки совершенства проектируемого двигателя.

Задачу выбора проектных параметров двигателя сформулируем как поиск рациональных значений вектора X , для которого выполняются условия технического задания (ТЗ). ТЗ определим как систему неравенств, регламентирующих предельные уровни наиболее важных показателей эффективности системы ГТД-ЛА и проектных ограничений по ГТД [41]:

$$\begin{cases} y_1(X, B) \leq y_{1\text{ТЗ}} \\ \dots \\ y_n(X, B) \leq y_{n\text{ТЗ}} \end{cases}, \quad (4.23)$$

где $y_{i\text{ТЗ}}$ заданный предельный уровень основных технических данных авиационного ГТД, который определяется в результате решения задачи поиска Парето-оптимальной оценки эффективности двигателя при прогнозируемых значениях исходных данных $\bar{B}$ (см. п. 4.1).

Фиксация значений исходных данных на уровне прогноза $\bar{B}$ и выбор соответствующего этому уровню эффективной оценки $Y_{\text{ТЗ}}$ целесообразны в том смысле, что в идеале, при совпадении запроектированных значений исходных данных с реально достигнутыми будет наблюдаться достижение эффективного сочетания целевых функций. Отклонение вектора B от прогноза $\bar{B}$ одинаково плохо как в случае превышения прогнозных значений, так и в случае недобора, поскольку в первом случае можно было бы найти более эффективную оценку целевых функций, а во втором - нарушатся условия ТЗ.

При выборе параметров сложных технических систем в условиях неопределенности исходной информации можно применить метод, основанный на определении *риска невыполнения* ТЗ [45,32,23].

Технический риск невыполнения заданных требований к проектируемой системе определим как вероятность того, что реализованные в процессе создания ГТД проектные характеристики выйдут за пределы, обусловленные техническим заданием:

$$R_T(X) = \int_{\bar{Z}(X)} \varphi_B(B) d\mu, \quad (4.24)$$

где $\bar{Z}(X) \in \Omega_B$ подмножество исходных данных, для элементов которого не выполняются условия ТЗ (4.23).

Задачу выбора проектных параметров авиационного двигателя можно сформулировать как поиск рационального вектора X , для которого выполняются условия технического задания (4.23). С учетом прогнозного характера исходной информации эту задачу можно уточнить, потребовав достижения наибольшего уровня вероятности выполнения условий (4.23), то есть минимального уровня технического риска

$$R_T(X) \rightarrow \min_X. \quad (4.25)$$

При формировании ТЗ методами, описанными в п.5.2., реализуется «раскрытие» критериальной неопределенности. Решение оптимизационной задачи (4.25) «раскрывает» неопределенность в исходных данных.

Для вычисления технического риска введем характеристическую функцию

$$\chi(B) = \begin{cases} 1, & \text{если } \forall i y_i(X, B) \leq y_{iTЗ}, \\ 0, & \end{cases} \quad (4.26)$$

тогда критерий технического риска можно вычислить по формуле

$$R_T(X) = \int_{\Omega_B} (1 - \chi(B)) \varphi_B(B) d\mu. \quad (4.27)$$

Исследуем свойства критерия технического риска.

Исходные предпосылки

Для проведения качественного анализа свойств критерия технического риска примем следующие упрощающие предположения:

1. Критерии оценки эффективности ГТД-ЛА $y_i(X, B)$ являются выпуклыми, унимодальными, непрерывными функциями по проектным параметрам и непрерывными, монотонно-убывающими функциями по любому направлению возрастания значений исходных данных b_i (данное допущение предполагает, что исходные данные имеют смысл коэффициентов полезного действия или могут быть так интерпретированы). Для модели ГТД такое допущение является вполне естественным.

2. Функции $y_i(X, B)$ неразделимы по переменным X и B

$$\forall X \in \Omega_X, \forall B \in \Omega_B (y_i(X, B) \neq \varphi_i(B)f_i(X) + C_i(X)) \quad (4.28)$$

3. В окрестностях любого проектного решения X_0 и вектора исходных данных B^* возможна линейризация критерия эффективности

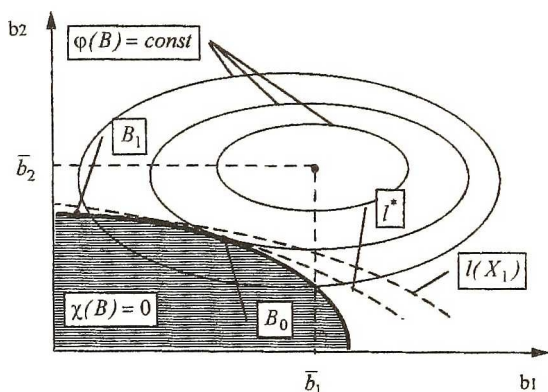


Рис.4.6' Оценка критерия технического риска

$$\forall X_0 \in \Omega_X, \forall B^* \in \Omega_B (y_i(X, B) \approx A_{iX_0B^*} B + C) \quad (4.29)$$

Для произвольного проектного решения X определим множество «благоприятных» исходов $Z(X)$ как совокупность таких векторов исходных данных B , для которых характеристическая функция равна 1, то есть

$$Z(X) = \{B | \chi(B) = 1\}. \quad (4.30)$$

Множество «неблагоприятных» $\bar{Z}(X)$ исходов определим как дополнение множества $Z(X)$ в пространстве исходных данных

$$\bar{Z}(X) = \Omega_B \setminus Z(X). \quad (4.31)$$

Определим понятие оболочки множества $Z(X)$ как ее границы: $l(X) = \text{Гр } Z(X)$. В множестве оболочек $\{l(X_\alpha)\}$; соответствующем множеству допустимых проектных решений $\{X_\alpha\}$ (для которых выполняются условия ТЗ), введем строгий и нестрогий порядок, положив

$$l(X_1) < l(X_2), \text{ если } (Z(X_2) \subset Z(X_1)) \& (l(X_1) \cap l(X_2) = \emptyset) \quad (4.32)$$

$$\text{и} \\ l(X_1) \leq l(X_2), \text{ если } (Z(X_2) \subset Z(X_1)).$$

Рассмотрим наиболее простой случай, когда ТЗ (5.23) содержит только один критерий: $y(X, B) \leq y_{ТЗ}$. Дополним множество оболочек $\{l(X_\alpha)\}$ минорантной оболочкой l^* :

$$l^* = \left\{ B \mid \min_X y(X, B) = y_{ТЗ} \right\}, \quad (4.33)$$

лежащей ниже любой оболочки множества $\{I(X_\alpha)\}$, в крайнем случае касающейся ее в одной точке (см. рис.4.6). Сформулируем без доказательства следующие утверждения.

Утверждение 4.3. Для принятых предположений 1-3 относительно свойств целевых функций $y_i(X, B)$, для любого проектного решения X_α справедливо $l^* \leq I(X_\alpha)$, то есть l^* - минимальный элемент множества выпуклых оболочек.

Утверждение 4.4. Существует взаимнооднозначное соответствие между проектным решением X^0 и вектором исходных данных B^* , определяемое условием $\min_X y(X, B^*) = y(X^0, B^*) = y_{T3}$.

Утверждение 4.5. Справедлива следующая оценка снизу для критерия технического риска

$$\min_X R_T(X) > \int_{\bar{Z}^*} \varphi_B(B) d\mu, \quad (4.34)$$

где $\bar{Z}^*$ - множество «неблагоприятных» исходов, имеющих оболочку l^* .

Лемма 4.1. Для замкнутых, выпуклых множеств $\{D_\alpha\}$ равного объема ($\mu(D_\alpha) = V = \text{const}$) и гауссовской функции плотности распределения $\varphi(B)$ максимум вероятности $\int_{D_\alpha} \varphi(B) d\mu$ будет заключен в эллипсоиде «рассеяния»

$$D_\lambda = \{B | \varphi(B) \leq \lambda(V)\}.$$

Лемму 4.1 приведем без доказательства, поскольку представляемый факт интуитивно понятен.

Утверждение 4.6. Для функции $y(X, B)$, удовлетворяющей предположениям 1-3 и функции плотности распределения, удовлетворяющей условиям леммы 1, существует единственная оболочка $I(X^0) \in \{I(X_\alpha)\}$ такая, что $R_T(X^0) = \min_X R_T(X)$.

Для краткости дадим геометрическое толкование утверждению 4.6

Из утверждения 4.5 следует, что нижняя оценка минимума критерия $R_T(X)$ определяется минорантной оболочкой l^* . Последняя формируется как касательная «поверхность» к семейству поверхностей $\{I(X_\alpha)\}$, разделяющих пространство исходных данных на две части (см. рис 4.6). Можно показать, что при прочих равных условиях (равенстве меры оцениваемого множества), вероятность больше для того множества, которое содержит в себе больший по диаметру эллипсоид рассеяния. Очевидно, что наибольший по диаметру эллипсоид содержит множество, ограниченное «поверхностью» $l(B^0)$, проходящей через точку касания минорантной оболочки l^* с эллипсоидом рассеяния (точка B^0 см. рис. 4.6). В соответствие с утверждением 4.4, каждой точке «поверхности» $B^* \in l^*$ соответствует единственный вектор проектных

параметров X^* . Так например, точке B^0 соответствует единственный вектор проектных параметров X^0 . Таким образом, в точке B^0 достигается максимум функции $\int_{\Omega_B \setminus Z(X^0)} \varphi(B) d\mu$ или минимум функции $\int_{Z(X^0)} \varphi(B) d\mu$, что соответствует

минимуму функции технического риска в точке X^0 . Утверждение 4.4 обеспечивает единственность минимума, поскольку любая другая оболочка семейства $I(X_\alpha)$ уменьшает вероятность соответствующего множества $Z(X)$ и увеличивает технический риск $R_T(X)$.

Полученные результаты легко обобщаются на случай многокритериальной задачи оптимизации, если вместо целевой функции $y(X, B)$ рассматривать «свертку» критериев $F(X, B) = \max_i y_i(X, B) / \Delta y_i^*$.

Несложно показать, что функция $F(X, B)$ удовлетворяет предположениям 1-3.

Полученные результаты позволяют сформулировать основные свойства критерия технического риска.

Наибольший интерес вызывает исследование зависимости критерия от вида закона распределения исходных данных. Дело в том, что построение более или менее точных статистических моделей распределения исходных данных невозможно как в силу прогностического характера формирования соответствующих оценок, так и в силу высокой стоимости получения в результате экспериментов, особенно в таких областях как проектирование ГТД. Обычно законы распределения исходных данных формируются методом экспертных оценок [32, 41]. Большая чувствительность критерия технического риска к виду закона распределения могла бы существенно сузить область его применения.

Особенности рассматриваемой предметной области не позволяют реализовать эффективное применение, развиваемых в последнее время, методов разработки программного обеспечения, например объектно-ориентированного программирования. Технологическая цепочка проектирования двигателя содержит большое количество взаимосвязанных проектных операций, например, для начальных этапов проектирования от формирования ТЗ на разработку двигателя, проектирования облика рабочего процесса, формирования конструктивно-геометрического облика ГТД до расчета характеристик двигателя, проектирования основных узлов и агрегатов, а также проектирования элементов узлов двигателя.

Математические модели ГТД или его составляющих компонент, необходимые для выполнения каждой проектной операции, могут существенно изменяться как по уровню сложности, так и по своей сути. Предварительная заготовка таких моделей под каждую конкретную задачу невозможна. В этих условиях необходимо обеспечить гибкое и оперативное формирование требуемых расчетных моделей из имеющихся в программном обеспечении.

Из утверждения 4.6 следует, что минимум технического риска формируется в точке касания эллипсоида рассеяния с минорантной оболочкой l^* . Как видно из рисунка 4.7, изменение вида закона распределения незначительно изменяет положение точки касания на поверхности l^* , что обеспечивает незначительное отклонение минимума критерия технического риска.

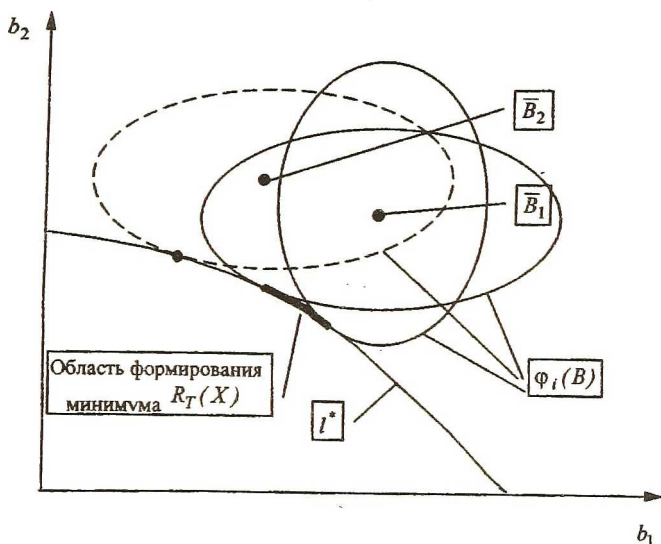


Рис. 4.7 Формирование минимума критерия $R_T(X)$ в зависимости от вида закона распределения исходных данных.

Значительно сильнее сказывается изменение положения ожидаемого прогнозного значения вектора исходных данных $\bar{B}$ (погрешность в задании центра закона распределения исходных данных) и то в плоскости параллельной поверхности l^* .

Очевидно, что пропорциональные изменения ковариационной матрицы закона распределения не окажут влияния на положение минимума критерия $R_T(X)$, изменив, однако, абсолютную величину критерия. В то же время следует отметить, что при выборе проектных параметров авиационного ГТД величина критерия технического риска не имеет существенного значения. Здесь технический риск применяется для поиска таких сочетаний параметров, которые обеспечили бы максимально возможную вероятность выполнения условий ТЗ.

Наиболее существенное влияние на положение минимума $R_T(X)$ оказывают условия ТЗ, поскольку изменение технического задания изменяют

не только положение поверхности l^* , но и (из утверждения 4.4) функциональные соотношения между векторами X и B .

Полученные результаты были подтверждены соответствующими вычислительными экспериментами [32, 23], проведенными на математической модели ТРДД в системе среднего магистрального самолета типа ТУ-204.

Исследование критерия технического риска показывает, что вид закона распределения исходных данных, а также его параметры, не могут оказать существенного влияния на значения проектных параметров, минимизирующих технический риск. Наиболее существенное влияние оказывает выбранная стратегия проектирования, выражающаяся в назначении как состава показателей, участвующих в формировании ТЗ, так и их предельных значений.

Вычисление критерия технического риска

Теоретически критерий технического риска можно вычислить используя формулы (4.24), (4.30) или (4.27). Однако для больших по размерности пространств исходных данных, в условиях отсутствия аналитических выражений описания множества $\bar{Z}(X)$, использование указанных выше формул становится невозможным.

Практичнее для вычисления $R_T(X)$ использовать метод Монте-Карло. В этом случае

$$\hat{R}_T(X) \approx \left[N - \sum_{i=1}^N \chi(B^{(i)}) \right] / N, \quad (4.35)$$

где $B^{(i)} \in \Omega_B$ — i -я реализация вектора исходных данных, распределенная в пространстве исходных данных в соответствии с плотностью распределения $\varphi(B)$.

Однако вычисление технического риска по формуле (4.35) требует большого числа обращений к математической модели двигателя. На базе утверждения 4.6 можно построить более эффективный алгоритм нахождения минимума критерия технического риска.

Алгоритм состоит из 4-х шагов.

Шаг 1. В направлении уменьшения модуля вектора исходных данных (например, направлении главной диагонали координатного угла) ищется точка пересечения луча, исходящего из точки $\bar{B}$ с минорантной оболочкой l^* (см. рис.4.8).

Шаг 2. В окрестности точки B^* функция $F(X, B)$ лианеризуется

$$F(X^*, B) \approx A_{X^*B} \cdot B + C. \quad (4.36)$$

Шаг 3. Используя построенную лианеризованную модель и функцию плотности распределения исходных данных, находим точку касания плоскости (4.36) с эллипсоидом рассеяния исходных данных, то есть точку B^{**} .

Шаг 4. Повторяются шаги 1-3 до тех пор, пока не будет найдена точка касания эллипсоида рассеяния с оболочкой l^* . Из утверждения 4.6 в этой точке формируется минимум критерия технического риска.

Предложенный алгоритм позволяет существенно сократить вычислительные затраты при поиске минимума критерия технического риска.

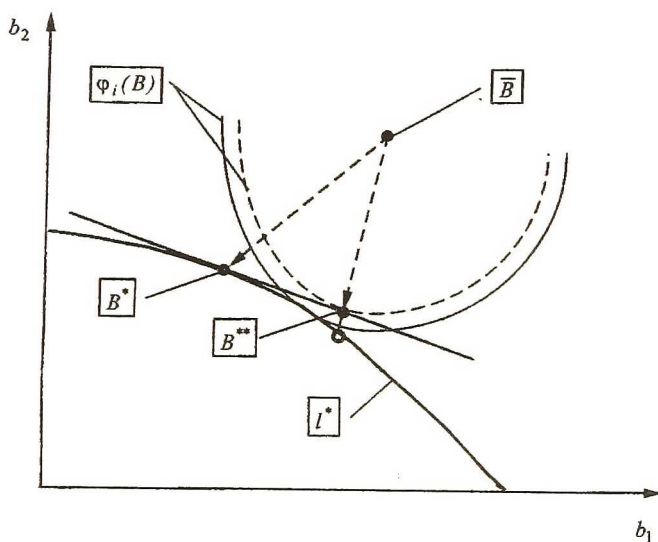


Рис.4.8 Поиск минимума критерия $R_T(X)$

4.1.4.Использование технологии ГСП для автоматизации разработки ПП начальных этапов проектирования авиационных ГТД

Начальный этап проектирования авиационных ГТД обычно связывают с выбором оптимальных для ЛА параметров СУ и его элементов на основе согласования характеристик двигателя и ЛА. На сегодня известны САПР ГТД, созданные в ЦИАМ (Г.В. Васильев, О.Д. Селиванов, О.К. Югов и др.), в МАИ (С.В. Румянцев, В.А. Сгилевский и др.), в СГАУ [30, 41]. Большинство работ в области САПР ГТД посвящено различным их модификациям. В то же время возрастающая сложность решаемых задач при создании двигателей нового поколения требует применения более детализированных и адекватных математических моделей двигателей, а также использования совершенных математических методов принятия решений и программного обеспечения, способного оперативно модифицироваться для решения меняющихся задач проектирования. В последнее время наметилась тенденция резкого возрастания затрат на разработку и модификацию программного обеспечения САПР, в связи с чем возросла значимость средств автоматизации разработки и модификации программного обеспечения.

Модель термогазодинамического расчета ТРДД.

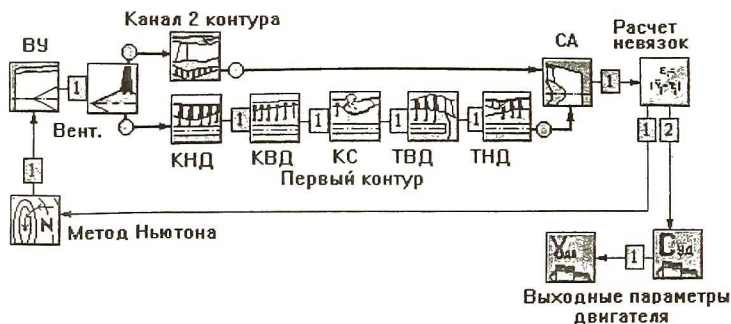


Рис. 4.9. Термогазодинамический расчет ТРДД.

Невозможность априорной подготовки всеобъемлющей совокупности математических моделей обуславливает три важных требования к САПР ГТД: система должна обладать свойством *гибкости*, которое позволяло бы проектировщику оперативно формировать необходимое программное обеспечение исходя из постановки конкретной задачи проектирования, свойством *открытости*, позволяющим легко дополнять имеющееся программное обеспечение новыми модулями, и *безболезненностью*, позволяющему эффективно модифицировать существующее программное обеспечение [30, 31, 41]. Перечисленным требованиям в полной мере удовлетворяет технология графо-символического программирования.

Так например, построение термогазодинамической модели авиационного двигателя основывается на принципе поузлового расчета составляющих его элементов. Причем структуризация модели двигателя зависит как от сложности рассматриваемого типа двигателя, так и от степени детализации проектно-конструкторской задачи. На рисунке 4.9 с помощью выразительных возможностей технологии ГСП представлена модель ТРДД.

В качестве узлов двигателя выделены следующие элементы: входное устройство (ВУ), вентилятор, компрессор низкого давления (КНД), компрессор высокого давления (КВД), камера сгорания (КС), турбина высокого давления (ТВД), турбина низкого давления (ТНД), сопловой аппарат (СА), канал второго контура. Расчет первого и второго контуров можно производить независимо друг от друга, поэтому на графе соответствующие ветки программы обозначены в параллельной схеме исполнения.

На этапе выбора параметров рабочего процесса ГТД модель двигателя представляется системой нелинейных уравнений, описывающих законы сохранения энергии, балансовые соотношения для рабочего тела, кинематические связи между узлами двигателя. В результате расчетов формируется вектор невязок $E=(e_1, e_2, \dots, e_n)$, минимизацию модуля которого обеспечивает модуль решения системы уравнений методом Ньютона. В результате термогазодинамического расчета вычисляются выходные характеристики двигателя, в том числе $\gamma_{дв}$ и $C_{уд}$.

Граф (рис. 4.9) полностью отражает структурную модель ТРДД, являясь при этом одновременно исчерпывающим описанием алгоритма расчета двигателя. Организация межмодульного информационного интерфейса была реализована ранее на этапе описания компонент графа (составляющих его объектов), путем привязки типов данных базовых модулей к данным предметной области. Как видно из рисунка, “текст” программы достаточно нагляден и напоминает иллюстративный материал, широко используемый в учебных пособиях.

Данная модель пригодна для использования на этапе проектирования рабочего процесса двигателя, однако для расчета высотно-скоростных характеристик ГТД необходим поступенчатый расчет компрессоров и турбин двигателя, что требует более детальной проработки моделей узлов ГТД. Изменение схемы расчета может быть вызвана вносимыми конструктивными изменениями в схеме двигателя. В технологии ГСП, при наличии богатой библиотеки моделей узлов и элементов ГТД, модификация программы реализуется за счет корректировки граф-программы, что не составляет труда даже для неквалифицированного в программировании специалиста..

Для расчета критериев оценки эффективности системы ГТД-ЛА, кроме основных выходных характеристик двигателя, привлекают дополнительную (чаще статистическую) информацию. Каждый из общепризнанных критериев оценки качества системы ГТД-ЛА может быть вычислен практически независимо друг от друга, что позволяет организовать параллельное вычисление критериев эффективности и снизить временные затраты на расчет авиационного двигателя в одной точке (см. рис. 4.10). Учитывая высокую трудоемкость расчетов модели ГТД и высокую сложность оптимизационных исследований, экономия времени играет не последнюю роль. Таким образом, возможность организации параллельных вычислений, предоставляемая технологией ГСП, не только повышает производительность создаваемого программного обеспечения, но и придает большую выразительность создаваемым граф-программам. Модификация схемы граф-программы не требует значительных затрат времени, следовательно, в зависимости от содержания проектной операции в модуль (рис. 4.10) можно включать только те объекты, которые участвуют в проектной операции, чем добиться еще большей производительности ПП.

и (4.19) оцениваются частные производные $\frac{\partial G(X)}{\partial y_i}$, $\frac{\partial^2 G(X)}{\partial y_i^2}$ и вычисляются градиент и гессиан “разделяющей” функции $G(X)$, т.е. формируется приближенная модель вида (4.13).

В процессе выработки технического задания на проект авиационного ГТД принимает участие ЛПР. Для выбора запроектированного уровня совершенства двигателя целесообразно, наряду с построением графика Парето-оптимального множества оценок, представлять ЛПР “опорные” эффективные оценки, полученные в соответствии с теми или иными принципами оптимальности. На рисунке 4.13 представлены три принципа выбора эффективных оценок из Парето-оптимального множества оценок. Принцип “равенства отклонений” исходит из предположения о равенстве приоритетов оптимизируемых критериев. В таких условиях из множества эффективных оценок следует выбирать такую точку, в которой все критериальные отклонения равны между собой.

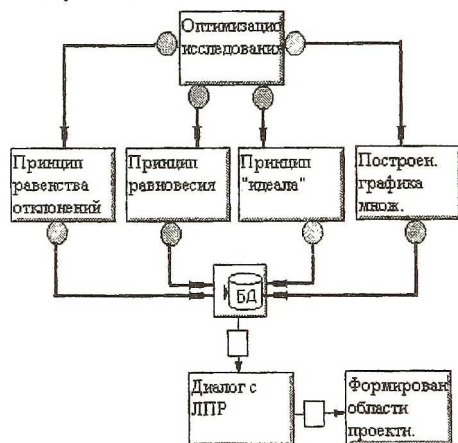


Рис. 4.13 Формирование технического задания

Принцип “равновесия” ищет такое сочетание критериальных отклонений, при котором улучшение ни одного из показателей эффективности не может быть реализовано за счет меньшего ухудшения значений других критериев.

Принцип “идеальной точки” осуществляет поиск эффективной оценки, ближайшей к соотношению критериев эффективности, предлагаемых ЛПР. Список принципов оптимальности может быть легко расширен, а соответствующая граф-программа скорректирована под новую задачу. Однако окончательное решение принимает ЛПР по результатам анализа таблицы

опорных эффективных оценок, исследования графического образа множества эффективных оценок с привлечением личной интуиции и опыта.

Обычно ТЗ формируется таким образом, чтобы область выбора проектных параметров не вырождалась в единственную точку. В этом случае появляется возможность учесть неформализуемые факторы ЛПР на этапе выбора проектного решения, которые невозможно применить на этапе выбора стратегии проектирования двигателя (формирования ТЗ). С этой целью производится расширение области проектных параметров за счет незначительных пропорциональных увеличений критериальных отклонений.

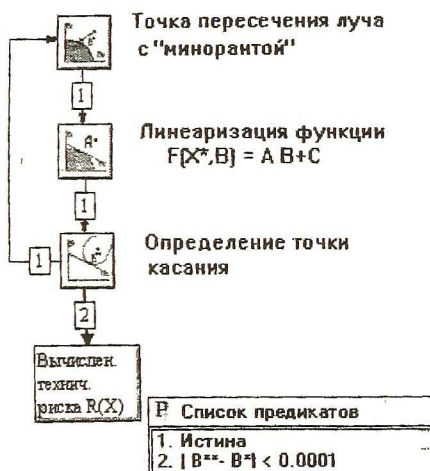


Рис.4.14 Минимизация технического риска.

Выбор проектного решения из области компромиссных решений отдельный этап проектирования двигателя, для реализации которого разработано достаточно большое количество методов. Достаточно подробно данная процедура описана в работах [41, 32, 28], там же можно найти библиографические ссылки на соответствующую литературу. Одним из подходов к выбору проектных параметров авиационного двигателя является его оптимизация по критерию технического риска [32, 23]. На рисунке 4.14 представлен алгоритм минимизации технического риска, теоретические основы которого описаны в п.4.1.3.

Представленная на рисунке программа исходит из ситуации, когда закон распределения исходных данных описывается совместной плотностью распределения типа гауссовского распределения. В тех случаях, когда область значений каждого из данных задается интервалами, а плотность распределения описывается равномерным законом распределения, представленный алгоритм необходимо модифицировать. В этом случае минимум критерия технического

риска формируется в точке пересечения перпендикуляра, опущенного из прогнозируемых значений исходных данных на минорантную оболочку. Представленная на рисунке 4.14 программа оптимизации технического риска позволяет найти соответствующие проектные параметры. Для вычисления значения минимума технического риска необходимо разрабатывать отдельную программу, содержание которой зависит от условий решаемой задачи.

Представленный в разделе материал никоим образом не исчерпывает всех многочисленных проблем, возникающих при проектировании авиационного двигателя или другой сложной технической системы. Теория и методы начальных этапов проектирования достаточно подробно описаны в работе [41]. Цель данного раздела заключалась в демонстрации возможностей технологии ГСП описывать программы в таких динамически изменяющихся областях, как построение процедур принятия решений и составления математических моделей авиационных ГТД.

4.2. Исследование погрешностей вычислительных алгоритмов

Разрабатываемая в качестве средства автоматизации разработки программного обеспечения САПР, технология ГСП, в первую очередь, ориентирована на описание вычислительных алгоритмов. Именно в этой сфере наиболее привычна и естественна процедурная парадигма программирования. Предложенный метод тестирования модулей также исходит из посылки о “численной” природе отлаживаемых модулей. В то же время любые алгоритмы численного анализа в условиях их реализации на ЭВМ оперируют двумя основными понятиями: сходимостью метода и его устойчивостью. Основной проблемой численных методов является разработка вычислительного алгоритма, удовлетворяющего требованиям *высокой точности, устойчивости и экономичности*.

Проблема точности численного метода определяется за счет решения задач оценки ошибки метода и оценки ошибок округления. Если в первом случае можно “вывести” аналитические формулы для оценки методических ошибок, применяя, например, методы теории разностных схем [40], то во втором случае получение аналитических оценок для реально используемых программ практически невозможно. Причин здесь несколько. Во-первых, невозможно проследить всех особенностей организации вычислений, реализованных в языке программирования и в операционной системе ЭВМ, во-вторых, любые оценочные формулы, полученные на основе теории погрешностей, являются слишком грубыми и приемлемы для оценки погрешностей сравнительно небольших формул. В то же время, ограниченность разрядной сетки ЭВМ и эффект распространения арифметических ошибок вызывают необходимость оценивать ошибки округления, особенно для сложных программ. Выход из сложившейся

ситуации видится в измерении ошибок округления непосредственно на объекте исследования - программном продукте.

В разделе рассматривается применение предлагаемого метода тестирования программных модулей в целях исследования сходимости численных схем решения систем нелинейных уравнений и оценки инструментальной погрешности ЭВМ.

4.2.1. Исследование сходимости численных методов решения системы нелинейных уравнений

Общепринятой практикой построения математических моделей ГТД (на начальных этапах проектирования двигателя) является использование системы нелинейных функциональных уравнений. Функции, входящие в уравнение, относятся к классу алгоритмических, в которых используются итерационные циклы, аппроксимации экспериментальных зависимостей, сложные функциональные связи. В связи с чем они не всегда являются гладкими, и даже непрерывными, функциями. Для решения систем уравнений, описывающих модель ГТД, обычно используют метод Ньютона или его модификации, поскольку последний обладает сверхлинейной сходимостью. Практический опыт решения данной задачи показал известную неустойчивость метода Ньютона. Попытки выяснения причин неудач не дали положительных результатов, поэтому была предложена схема дублирования метода Ньютона другими методами решения систем нелинейных уравнений с организацией перехвата программных прерываний. Методы тестирования программных модулей, описанные в главе 3, позволили выяснить причины неустойчивой работы метода Ньютона.

Исследованию была подвергнута достаточно простая система нелинейных уравнений, решение и исследование которой не вызывает технических трудностей:

$$\begin{cases} \sin^2 x + \cos^2 y - 1 = 0, \\ y - \ln x = 0. \end{cases} \quad (4.37)$$

Первое уравнение системы не является гладкой функцией, ее производная в некоторых точках терпит разрывы первого рода. Поскольку для системы (5.37) предоставляется возможность вывести аналитические выражения для вычисления первых частных производных, то первоначально рассматривался "чистый" алгоритм метода Ньютона (теоретически наиболее быстрая схема решения системы нелинейных уравнений).

На рисунке 4.15 представлена структура области расходимости метода Ньютона для системы (4.37). Светлые зоны соответствуют случаю расходимости алгоритма решения системы уравнений, темные его сходимости. Рисунки б) и в) соответствуют пропорциональному увеличению обозначенных зон. Как видно из рисунка, область сходимости метода Ньютона,

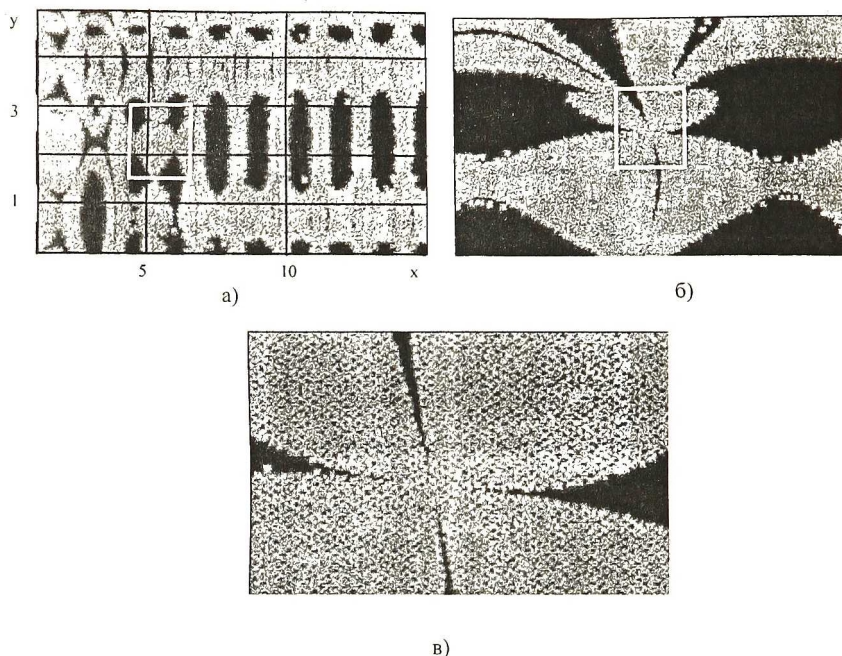


Рис.4.15. Структура области расходимости метода Ньютона в случае аналитического вычисления производных

даже для такого простого примера, имеет нетривиальную структуру, а метод Ньютона расположен в большей степени к, нежели чем к сходимости.

Для функций, описывающих модель двигателя, аналитическое вычисление их частных производных невозможно, для этих целей обычно используют конечно-разностные схемы. На рисунке 4.16 показана область расходимости метода Ньютона в случае численного способа вычисления производных.

Из рисунка видно, что область сходимости метода Ньютона, как это ни парадоксально, значительно увеличилась, а структура области приняла “линейчатый” характер даже в зонах расходимости алгоритма. “Линейчатость” зон расходимости можно объяснить ограниченностью разрядной сетки ЭВМ и дискретной природой конечно-разностных схем вычисления производных. Как видно из рисунка, для обеспечения сходимости метода достаточно лишь незначительно изменить начальное приближение метода.

Влияние изменения шага конечно-разностной схемы вычисления производных на сходимость метода показано на рисунке 4.17.

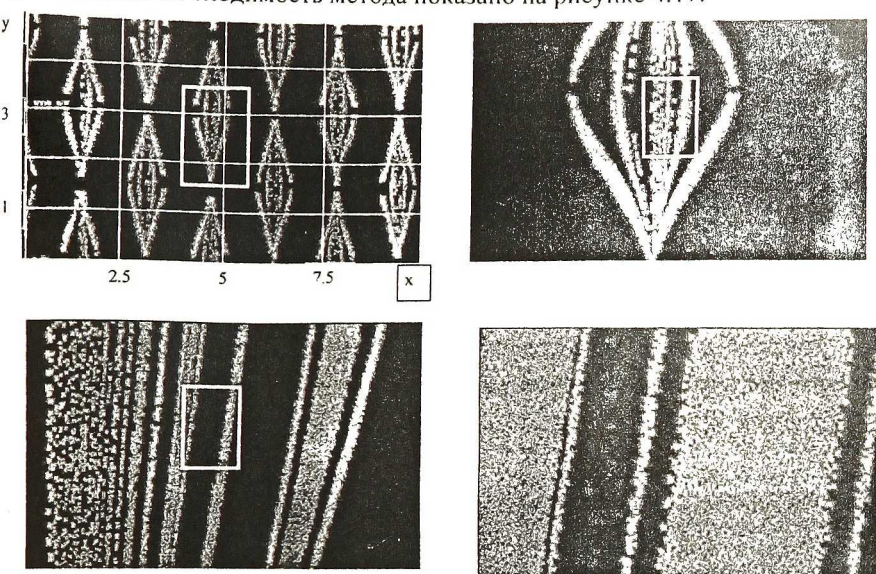


Рис.4.16. Структура области расходимости метода Ньютона в случае численного вычисления производных ($h=10^{-11}$)

Как видно из рисунка, грубое задание шага (высокие методические погрешности) ухудшает сходимость алгоритма. В то же время приближенный способ вычисления производных (по сравнению с аналитическим) улучшает сходимость метода.

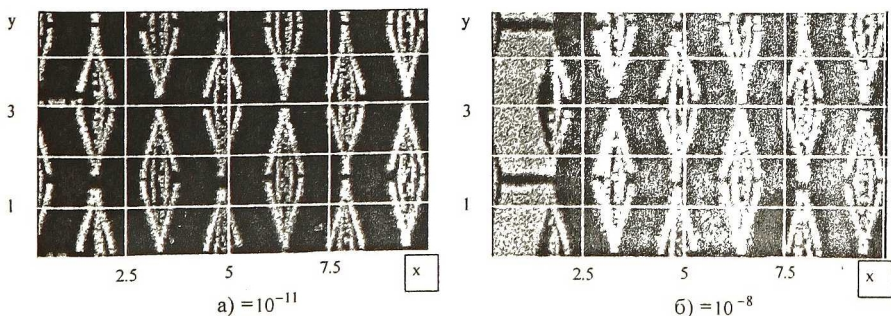


Рис.4.17 Зависимость структуры области расходимости метода Ньютона от шага вычисления производных

Проведено теоретическое исследование сходимости представленных вариантов метода решения системы уравнений на основе достаточного признака сходимости итерационных методов. Сходимость оценивалась по признаку $\rho(G'_X(X)) < 1$, где $\rho(\cdot)$ - спектральный радиус производной итерационной функции $G'_X(X)$. Для метода Ньютона с аналитическим способом вычисления производных пример позволяет произвести исследование глобальной сходимости, рассматривая якобиан как матричную функцию от независимых переменных $G(X) = X - J^{-1}(X)F(X) = X - \Psi(X)$. В случае численного способа вычисления частных производных на каждой итерации алгоритма якобиан представлялся численной матрицей J^{-1} , в связи с чем итерационная функция рассматривалась в виде $G(X) = X - J^{-1}F(X)$. На рисунке 4.18 представлены "портреты" областей расходимости для рассматриваемых случаев: а) аналитическое вычисление производных, б) численное вычисление производных.

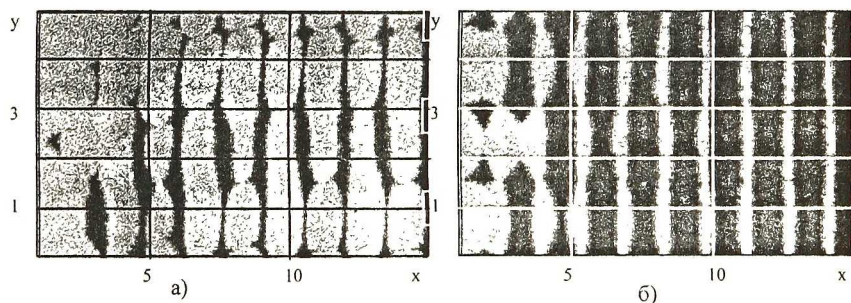


Рис 4.18 Области расходимости метода Ньютона

Как видно из рисунка, теоретические области расходимости отличаются от практически реализуемых областей расходимости (см. рис. 4.15 а), 4.17 а) и 4.18). Последнее может означать лишь тот факт, что дискретная природа ЭВМ, как вычислительного средства, серьезным образом сказывается на реализации континуальных моделей. В то же время следует отметить, что негладкость функций, входящих в систему уравнений, существенно ограничивает возможное применение метода Ньютона, даже в теоретическом плане.

На рисунке 4.19 продемонстрирован "портрет" области сходимости метода деформированных многогранников, использованный для решения той же задачи. Как видно из рисунка, метод сходится практически на всей области определения системы уравнений. Метод деформированных многогранников относится к классу методов, не использующих производных. В определенном смысле его можно считать методом дискретной математики. Полученные результаты наводят на мысль, что существует некоторый эффект взаимного

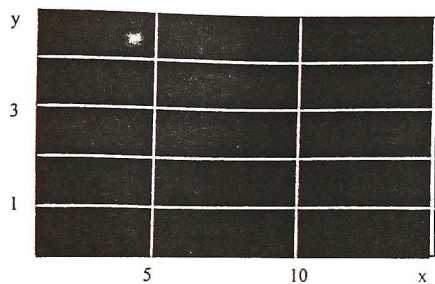


Рис.4.19 Область сходимости метода деформируемых многогранников

влияния между свойствами модели объекта исследования и методами его исследования. Так например, если модель имеет дискретную природу, (не является непрерывной или не имеет непрерывных производных), то для ее исследования лучше применять также дискретные методы. Из чего следует, что для решения систем нелинейных уравнений, описывающих модель ГТД, рациональнее применить метод деформированного многогранника.

4.2.2. Оценка инструментальной погрешности ЭВМ

Предложенный в п.3.1 метод искусственного расширения области ошибок можно использовать для оценки инструментальной погрешности численного алгоритма, возникающей в связи с накоплением ошибок арифметических операций.

Пусть $y = AX$ - некоторый численный метод, реализующий вычисление величины y (скалярной или векторной), где оператором A формально обозначен произвольный алгоритм численного метода, X исходные данные и переменные алгоритма. Очевидно, что в силу ограниченности разрядной сетки ЭВМ результат вычислений зависит от размера разрядной сетки ЭВМ t . В общем случае, в предположении, что точность задания исходной информации алгоритма зависит только от размера разрядной сетки ЭВМ, для оценки относительной погрешности имеем:

$\delta_y(t) = K\delta(t)$, где $\delta(t) = 10^{-t+1}$ относительная погрешность представления чисел в ЭВМ. Абсолютная погрешность может быть оценена по формуле $e_y(t) = Ky\delta(t)$. (4.38)

Оценить абсолютные ошибки округления алгоритма можно, например, построив граф вычислительного процесса, и на его основе вывести формулу

оценки ошибки округления [39], однако в общем случае проведение таких исследований невозможно.

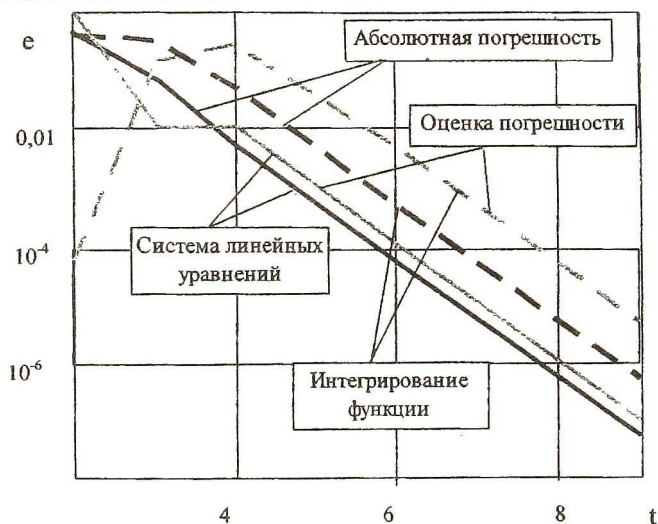


Рис.4.20 Зависимость ошибок округления e от размера разрядной сетки ЭВМ t на примерах решения системы линейных уравнений и интегрирования функции.

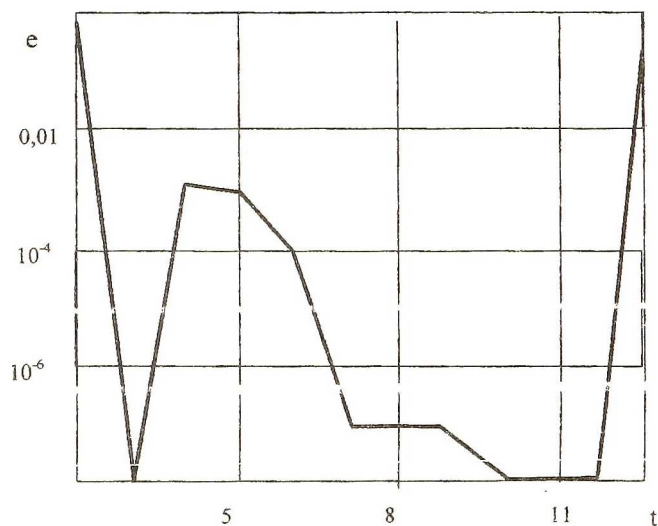


Рис.4.21 Зависимость ошибок округления e от размера разрядной сетки ЭВМ t для вырожденной системы линейных уравнений

Рассмотрим метод, основанный на искусственном ограничении разрядной сетки ЭВМ. Из формулы (4.38) видно, что для вычисления оценки $e_y(t)$ необходимо оценить величину параметра K . Для разных значений параметра t имеем:

$$\begin{aligned} y(t) &= y + e_y(t) = y + K\delta(t), \\ y(t+1) &= y + e_y(t+1) = y + K\delta(t+1). \end{aligned} \quad (4.39)$$

Из (2) имеем

$$|\Delta y(t)| = |\Delta e_y(t)| = |Ky| \cdot |\delta(t+1) - \delta(t)| = |Ky| \frac{9}{10} 10^{-t+1} \quad (4.40)$$

Из формулы (5.38) видно, что $e_y(t)$ линейно зависит от параметра K , тогда из (4.40) следует, что для оценки K требуется два вычисления y при разных значениях t . На рисунке 4.20 представлены результаты вычислительных экспериментов по оценке инструментальных погрешностей для двух моделей:

1. Решения системы линейных уравнений

$$\begin{cases} 2x_1 + x_2 + 3x_3 = 9 \\ x_1 - 2x_2 + x_3 = -2 \\ 3x_1 + 2x_2 + 2x_3 = 7 \end{cases}$$

при этом точность оценивалась для обобщенного параметра

$$\Delta y = |x_1 - x_1(t)| + |x_2 - x_2(t)| + |x_3 - x_3(t)|.$$

2. Вычисления определенного интеграла вида

$$\int_0^1 \frac{x^2}{(x^2 + 1/9)^2} dx.$$

Из рисунка 4.20 видно, что при малых значениях параметра t наблюдается потеря устойчивости алгоритма, однако, начиная с некоторого t , зависимость становится линейной. Для оценки параметра K можно предложить метод среднеквадратической аппроксимации результатов наблюдений y в m экспериментах при различных t на линейном участке изменения функции:

$$\beta = \lg K = \frac{1}{m} \sum_{t=n}^{n+m} [\lg(10 \cdot |\Delta y(t)| / 9 \cdot |y|) - (1 - t)] \quad (4.41)$$

Вычисленное по результатам экспериментов при различных t значение K подставляется в формулу (4.38), с помощью которой оценивается инструментальная погрешность исследуемого численного метода. Характер поведения ошибок округления в зависимости от разрядности ЭВМ t позволяет косвенно судить об устойчивости численной схемы вычисления. На рисунке 4.21 показана зависимость изменения ошибок округления от t для системы линейных уравнений с плохо обусловленной матрицей. Как видно из рисунка, наблюдается сильная неустойчивость результатов вычислений.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Архангельский Б.В., Черняховский В.В. Поиск устойчивых ошибок в программах. - М.: Радио и связь, 1989.
2. Ахо А., Хопкрофт Дж., Ульман Дж. Построение и анализ вычислительных алгоритмов. - М.: Мир, 1979.
3. Бозм Б., Браун Дж., Каспар Х. и др. Характеристики качества программного обеспечения. - М.: Мир, 1981.
4. Братко И. Программирование на языке ПРОЛОГ для искусственного интеллекта. - М.: Мир, 1990.
5. Вагин В.Н. Дедукция и обобщение в системах принятия решений. - М.: Наука, 1988.
6. Вельбицкий И.В., Ершов С.В. Графовая алгебра процессов и высокоуровневая отладка программ. УСиМ, №6, 1991.
7. Вентцель Е.С. Исследование операций М.: Сов. радио, 1972.
8. Вирт Н. Систематическое программирование. Введение. М.: Мир, 1977.
9. Виттих В.А., Коварцев А.Н., Кораблин М.А. Имитация автоматизированных систем с использованием концепции состояний. - М.: Известия АН СССР, Техническая кибернетика, 1981, №4.
10. Вязгин В.А., Федоров В.В. Формирование облика сложного технического объекта в САПР. // Программные продукты и системы. - 1990, №4.
11. Гермейер Ю.Б. Введение в исследование операций. - М.: Наука, 1971.
12. Гласс Р. Руководство по надежному программированию. М.: Финансы и статистика, 1982.
13. Глушков В.М. Основы безбумажной информатики. - М.: Наука, 1987.
14. Горбатов В.А. Основы дискретной математики М.: Высшая школа, 1986.
15. Горбунов-Посадов М.М. Конфигурации программ. - М.: Малип, 1994.
16. Дал У.И. Языки для моделирования систем с дискретными событиями. // В сб. "Языки программирования". М.: Мир, 1972.
17. Дейкстра Э. Дисциплина программирование. - М.: Мир, 1978.
18. Евстигнеев В.А. Применение теории графов в программировании. М.: Наука, 1985.
19. Закс Ш. Теория статистических выводов. - М.: Мир, 1975.
20. Замулин А.В. Системы программирования баз данных и знаний. - Новосибирск: Наука, 1990.
21. Ковалев А.Л., Дробушев Л.Ф., Ленчик Е.П. Оценка топологической сложности графических программ и генерация маршрутов для их тестирования. // УСиМ, 1990, №3.
22. Коварцев А.Н. Численный метод определения многомерных областей для выбора оптимальных параметров авиационных ГТД с помощью

- ЭВМ.// Сб.: Проектирование и доводка газотурбинных двигателей. Куйбышев, 1983.
23. Коварцев А.Н. Оценка критерия технического риска при выборе параметров рабочего процесса авиационного ГТД. Изв. вузов. Авиационная техника, №3, 1989.
 24. Коварцев А.Н. Численные методы.// Учебн. пособие. - Самарский муниципальный комплекс непрерывного образования. Самара, 1997.
 25. Коварцев А.Н., Боргест Н.М. Применение средств автоматизации программирования при разработке диалоговой подсистемы "Выбор параметров ГТД". // Сб. "Применение ЭВМ в научных исследованиях. М., 1986.
 26. Коварцев А.Н., Кудрин К.А. Метод классификации данных программных модулей, реализованных в технологии графосимволического программирования // Научно-методическая конференция "Интенсивные технологии обучения в подготовке специалистов" / Тез.докл. - Самара, 1996.
 27. Коварцев А.Н., Кудрин К.А. Технология графосимволического программирования в системе GRAF. // Тр. Международной конференции "Математика. Компьютер. Образование". - М.: РИИС ФИАН, 1996.
 28. Коварцев А.Н., Кузьмичев В.С. Автоматизированный метод построения областей компромиссов в задачах выбора параметров рабочего процесса ГТД.- Куйбышев, 1980. Рук.деп. в ВИНТИ 18 марта 1981. №1210-81
 29. Коварцев А.Н., Кузьмичев В.С., Иванов А.М. Пакет прикладных программ приближения функций с помощью оптимальных сплайнов. - УСиМ, 1986, №2.
 30. Коварцев А.Н., Кузьмичев В.С., Маслов В.Г. и др. Концепция построения и реализации гибкой САПР газотурбинных двигателей.- Куйбышев, 1988. Рук.деп. в ЦНИИТЭИ 27.09.88г №231ТМ88.
 31. Коварцев А.Н., Ломакин В.Б., Архипов А.М., Маслов В.Г. Реализация технологии модульного программирования инструментальной системы ТЕМП.// Куйбышев, 1989. Рук.деп. в ВИНТИ №1961-В89.
 32. Коварцев А.Н., Маслов В.Г. Оценивание технического риска на начальном этапе проектирования ГТД. - Куйбышев, 1988. Рук.деп. в ВИНТИ 6 декабря 1988. № ДД1335
 33. Кораблин М.А. Конструирование динамических взаимодействий в объектно-ориентированных системах. М.: АН СССР, Программирование, 1990, №6.
 34. Кудрин К.А., Коварцев А.Н., Прохоров С.А. Методы автоматизации отладки в технологии графосимволического программирования // Сборник научных трудов "Информационные системы и технологии", - Самара, 1996.

35. Липаев В.В. Отладка сложных программ. Методы, средства, технология. - М.: Энергратомиздат, 1993.
36. Лотов А.В. Введение в экономико-математическое моделирование. - М.: Наука, 1984.
37. Майерс Г. Искусство тестирования программ. - М.: Финансы и статистика, 1982.
38. Майерс Г. Надежность программного обеспечения. - М.: Мир, 1980.
39. Мак-Кракен Д., Дорн У. Численные методы и программирование на ФОРТРАНе. - М.: Мир, 1977.
40. Марчук Г.И. Методы вычислительной математики. - М.: Наука, 1980.
41. Маслов В.Г., Кузьмичев В.С., Коварцев А.Н., Григорьев В.А. Теория и методы начальных этапов проектирования авиационных ГТД// Учебн. пособие. - Самар. гос. аэрокосм. ун-т. Самара, 1996.
42. Мейер Б., Бодуэн К. Методы программирования: В 2-х томах. Т.1. - М.: Мир, 1982.
43. Мейер Б., Бодуэн К. Методы программирования: В 2-х томах. Т.2. - М.: Мир, 1982.
44. Нормативные и дискрептивные модели принятия решений.//По материалам советско-американского семинара. - М.: Наука, 1984.
45. Осин М.И. Методы автоматизированного проектирования летательных аппаратов. М.: Машиностроение, 1984.
46. Подиновский В.В., Ногин В.Д. Парето-оптимальные решения многокритериальных задач. - М.: Наука, 1982.
47. Райфа Г. Анализ решений. - М.: Наука, 1977.
48. Роджерс Х. Теория рекурсивных функций и эффективная вычислимость. - М.: Наука, 1972.
49. Современное состояние теории исследования операций. - М.: Наука, 1979.
50. Статистические модели и многокритериальные задачи принятия решений. - М.: Статистика, 1979.
51. Стернс Т., Стернс Л. Изучи сам Visual FoxPro 3.0... сегодня. - Минск: Попурри, 1997.
52. Тейер Т., Липов М., Нельсон Э. Надежность программного обеспечения. - М.: Мир, 1981.
53. Турский В. Методология программирования. - М.: Мир, 1981.
54. Успенский В.А., Семенов А.Л. Теория алгоритмов: основные открытия и приложения. - М.: Наука, 1987.
55. Cardelli L, Wegner P. On undstanding types, data abstraction and polymorphism // Computing Surveys. - 1985. - Vol. 17, N 4. - P. 471-522.
56. Dijkstra E.W. The Humble Programmer (ACM Turing Awards Lecture). CommACM, Oct.1972. P.859-866.
57. Kleene S.C., Post E.L. The upper semi-lattice of degrees of recursive unsolvabilite // Anals of mathematics. Ser. 2. 1954.-V.59, №3. -P 379-

407.

- 58.. McCabe T.J. A complexity measure // IEEE - Transactions on Software Engineering, - V9, 1976. - N4, - pp. 308-320
- 59. Parnas P. L. On Crireria to Be Used in Decomposing System into Modules // Comm. ACM. - 1972. - Voi. 15, № 12. - P. 1053-1058.

Научное издание

Коварцев Александр Николаевич

**АВТОМАТИЗАЦИЯ РАЗРАБОТКИ И ТЕСТИРОВАНИЯ
ПРОГРАММНЫХ СРЕДСТВ**

Компьютерный набор и верстка А.Н. Коварцев

Лицензия ЛР № 020301 от 30.12.96 г.

Подписано в печать 20.04.99 г. Формат 60х84 1/16.

Бумага газетная. Печать офсетная.

Усл. печ. л. 8,8. Усл. кр.-отт. 8,9. Уч.-изд.л. 9,5.

Тираж 1000 экз. Заказ 89.

**Самарский государственный аэрокосмический
университет им. академика С. П. Королева.
443086 Самара, Московское шоссе, 34.**

**ИПО Самарского государственного
аэрокосмического университета.
443001 Самара, ул. Молодогвардейская, 151.**