

**БИБЛИОТЕЧКА
ПРОГРАММИСТА**

М. М. ГОРБУНОВ - ПОСАДОВ
Д. А. КОРЯГИН
В. В. МАРТЫНЮК

Системное обеспечение пакетов прикладных программ



БИБЛИОТЕЧКА ПРОГРАММИСТА

М.М. ГОРБУНОВ-ПОСАДОВ
Д.А. КОРЯГИН, В.В. МАРТЫНЮК

СИСТЕМНОЕ ОБЕСПЕЧЕНИЕ ПАКЕТОВ ПРИКЛАДНЫХ ПРОГРАММ

Под редакцией А.А. САМАРСКОГО



МОСКВА "НАУКА"

ГЛАВНАЯ РЕДАКЦИЯ
ФИЗИКО-МАТЕМАТИЧЕСКОЙ ЛИТЕРАТУРЫ

1990

Г о р б у н о в - П о с а д о в М. М., К о р я г и н Д. А., М а р т ы н ю к В. В. под ред. А. А. Самарского
Системное обеспечение пакетов прикладных программ. - М.: Наука. Гл. ред. физ.-мат. лит., 1990. - 208 с. - (Библиотечка программиста.). - ISBN 5-02-014386-3.

Описываются архитектура и методы реализации системного обеспечения развитых пакетов прикладных программ. Пакет прикладных программ в настоящее время является одной из основных форм специализированного программного обеспечения и представляет собой комплекс взаимосвязанных прикладных программ и средств системного обеспечения (программных и языковых), предназначенный для автоматизации решения определенного класса задач.

Предметными областями для описываемых пакетов являются, как правило, различные классы задач математической физики, однако основные излагаемые системные решения могут быть применены и при других разнообразных предметных ориентациях пакетов.

Рассматриваемые в книге пакеты программ успешно используются в Институте прикладной математики имени М.В.Келдыша АН СССР и в других организациях для решения важных научных и инженерно-технических задач.

Для программистов. Может быть полезна студентам вузов.

Ил. 14. Библиогр. 140 назв.

Г 1404000000-054 162-90
053(02)-90

ISBN 5-02-014386-3

© Издательство «Наука»
Главная редакция
физико-математической
литературы, 1990

ОГЛАВЛЕНИЕ

Предисловие редактора	5
Введение	7
Г л а в а 1. Архитектура пакета прикладных программ	24
1.1. Системные характеристики пакета	24
1.2. Язык заданий	25
1.3. Пакетный подход к модуляризации программ	27
1.4. Регламент модуляризации функционального наполнения	32
1.5. Конфигурации модулей	35
1.6. Планирование вычислений	39
1.7. Операционные возможности	46
1.8. Информационное обеспечение	52
1.9. О последующих главах	56
Г л а в а 2. Программирование задач вычислительного эксперимента	57
2.1. Вычислительный эксперимент	57
2.2. Цикл вычислительного эксперимента. Обработка результатов	58
2.3. Подготовка расчетных программ	60
2.4. Проект OLYMPUS	63
Г л а в а 3. Система Сафра: конструирование программ вычислительного эксперимента	67
3.1. Технология конструирования	67
3.2. Организация и содержание работ по созданию Сафры	70
3.3. Архитектура	74
3.4. Архив	77
3.5. Дисциплина работы	78
3.6. Язык заданий	83
3.7. Операции над модулями	85
3.8. Сборка и запуск расчетного варианта	94
3.9. Вспомогательные операции	104

3.10.	Документирование	105
3.11.	Операции над архивом	107
Г л а в а 4 Система ПНФ и пакет ЗАЩИТА . . .		111
4.1.	Конструирование программ из нестандартизированных модулей	111
4.2.	Основные понятия системы ПНФ	113
4.3.	Характеристики параметров модуля	116
4.4.	Описание модуля	118
4.5.	Преобразователь	121
4.6.	Фрагменты расчетных цепочек	123
4.7.	Проведение расчета	126
4.8.	Пакет ЗАЩИТА	128
Г л а в а 5. Анализатор PLAN: обработка данных физических экспериментов		130
5.1.	Архитектура	130
5.2.	Общая характеристика анализатора PLAN-БЭСМ-6	136
5.3.	Разработка пакетов на основе анализатора PLAN-БЭСМ-6	146
5.4.	Внешнее информационное обслуживание	149
Г л а в а 6. Пакеты для конкретных приложений		153
6.1.	Пакет КРИТ. многокритериальная оптимизация	153
6.2.	Пакет ОКС	162
6.3.	Пакет МП: автоматическое планирование вычислений	168
Заключение		190
Словарь терминов		191
Список литературы		194

ПРЕДИСЛОВИЕ РЕДАКТОРА

Создание проблемно-ориентированных комплексов (систем), называемых пакетами прикладных программ, является важным направлением работ в вычислительной математике. Характерная особенность пакета программ состоит в том, что он может постоянно расширяться и развиваться благодаря включению новых модулей. При создании пакетов, помимо разработки и отбора включаемых в них алгоритмов и прикладных программ, большое значение имеют работы по соответствующему системному обеспечению. Быстрота и удобство решения конкретных классов задач при использовании пакетов достигаются сочетанием в единой архитектуре функционального наполнения, состоящего из модулей и покрывающего определенную предметную область, и специализированных средств системного обеспечения, позволяющих сравнительно легко реализовывать различные задания и обеспечивающих пользователя разнообразным сервисом при подготовке и прохождении задач.

Особый интерес представляет применение пакетного подхода к программированию задач вычислительного эксперимента. В этих задачах анализируются свойства некоего объекта посредством проведения серии расчетов над его математической моделью. В ходе исследования модель объекта претерпевает многочисленные изменения, которые неизбежно влекут за собой изменения соответствующих расчетных программ. В настоящее время пакет является практически единственной приемлемой формой организации таких программ, позволяющей удержаться на плаву в безбрежном море версий и вариантов исходной модели. Пакетная организация программ вычислительного эксперимента дает возможность систематизировать выполнение исследований, способствуя тем самым обеспечению достоверности получаемых результатов.

Предлагаемая книга познакомит читателя с архитектурой и методами реализации системного обеспечения развитых пакетов прикладных программ. Она представляет собой итог обширного цикла исследований проблематики системного

обеспечения пакетов. Рассматриваемые в книге пакеты программ успешно используются в Институте прикладной математики имени М.В.Келдыша АН СССР и в других организациях для решения важных научных и инженерно-технических задач.

Книга будет полезна для широкого круга специалистов, занимающихся или интересующихся прикладным или системным программированием. По представленному в ней материалу ее авторы на протяжении многих лет читают курс лекций студентам факультета вычислительной математики и кибернетики МГУ.

Академик А. А. Самарский

ВВЕДЕНИЕ

Одной из важных проблем в области использования вычислительной техники является проблема общения человека с вычислительной машиной при решении различных прикладных задач. Для повышения эффективности такого общения требуются постоянные усилия в области разработки и совершенствования соответствующих алгоритмов и программных средств. Здесь можно выделить три направления работ:

- создание программных средств, обеспечивающих пользователей различными инструментами для автоматизации разработки программ;
- создание программных средств, упрощающих процесс эксплуатации машин инженерно-диспетчерским персоналом, а также обеспечивающих эффективное использование всех ресурсов вычислительной системы;
- создание программных средств, предоставляющих пользователям разнообразные «вычислительные услуги» при решении прикладных задач.

Эти три направления сводятся к повышению соответственно уровней инструментальной, исполнительской и тематической квалификации вычислительной машины.

Длительное время основное внимание системного программирования было сосредоточено на вопросах общего программного обеспечения, связанных с решением задач повышения уровня инструментальной и исполнительской квалификации вычислительных машин. Это объясняется двумя тенденциями, имевшими место в развитии вычислительной техники. Первая из них - быстрый рост парка и расширение сферы использования вычислительных машин, что потребовало создания ряда универсальных систем автоматизации программирования, позволяющих более производительнее вести разработку и отладку программ решения прикладных задач. Особенно интенсивно работы в этом направлении велись в период шестидесятих годов.

Другая тенденция - это совершенствование возможностей, качественное усложнение организации и режимов рабо-

ты аппаратной части вычислительных систем. Здесь потребовалось создать развитые операционные системы, организующие обслуживание задач, обеспечивающее эффективное использование оборудования, программ и данных. Период интенсификации работ в этом направлении связан с появлением машин с новой архитектурой и приходится на 1965–1975 годы.

Что же касается задачи повышения уровня тематической квалификации вычислительных машин, то хотя на начальной стадии системное программирование почерпнуло из этой области постановки таких своих фундаментальных проблем, как трансляция, отладка и т.п., в дальнейшем оно ослабило свой интерес к непосредственному решению вопросов, связанных с обеспечением «вычислительных услуг». Повышение уровня тематической квалификации вычислительных систем осуществлялось главным образом коллективами прикладного программирования, создававшими алгоритмы и программы для решения задач в различных приложениях.

В течение ряда лет недостаточное внимание со стороны системного программирования к задаче повышения уровня тематической квалификации вычислительных систем серьезно не сказывалось на эффективности проведения вычислительных работ. Это объясняется тем, что одна из главных задач системного программирования – создание развитых средств формального описания алгоритмов решения задач и аппарата перевода таких описаний на язык машины – была с самого начала поставлена в связи с необходимостью программирования прикладных задач. Как мы уже отмечали, аналогичное происхождение имеет и ряд других задач системного программирования. Такой изначальный учет основных потребностей практики способствовал тому, что, несмотря на ограниченность связей между работами, выполняемыми прикладными и системными программистами, а также общецелевой характер штатных средств системного обеспечения, эти средства в достаточной мере удовлетворяли все системные запросы программистов-прикладников. В тех же случаях, когда оказывалось, что имеющихся системных средств недостаточно для какой-то прикладной разработки, соответствующие системные возможности реализовывались прикладниками самостоятельно, причем, как правило, дело сводилось к модификации (в той или иной степени) наличных системных средств.

В середине семидесятых годов эта ситуация претерпела существенные изменения. Штатное системное обеспечение вычислительных машин настолько усложнилось, что даже

квалифицированное использование всех предоставляемых им возможностей, а тем более работы по его модификации или развитию потребовали от пользователей весьма высокого уровня профессиональных знаний. Оказалось, однако, что общецелевые средства системного обеспечения не всегда адекватны тем новым формам ведения вычислительных работ и тенденциям их развития, которые наметились в большинстве приложений.

Эти негативные обстоятельства, затормозившие использование вычислительной техники во многих важных приложениях, явились следствием недостаточно согласованного подхода со стороны прикладного и системного программирования к тематическому аспекту проблемы общения человека с машиной.

В связи с этим на современном этапе развития средств организации общения человека с машиной особое внимание уделяется новым, более совершенным чем прежде формам решения задачи повышения уровня тематической квалификации вычислительных систем. Конкретно это означает, что одной из важных целей программирования стала разработка «дружественных» средств, обеспечивающих пользователям более удобный доступ к «вычислительным услугам», предоставляемым машиной, и требующих от них минимальной профессионально-программистской подготовки.

Повышение уровня тематической квалификации вычислительных систем осуществляется путем создания специализированных компонентов программного обеспечения, ориентированных на отдельные приложения [1-11]. При этом полнота и эффективность решения задачи повышения квалификации системы достигается за счет учета специфики прикладной деятельности, для которой предназначен разрабатываемый компонент программного обеспечения. Принципиальным моментом в организации работ над такими специализированными компонентами программного обеспечения является совместное участие прикладных и системных программистов на всех этапах разработки. Такой подход позволяет не только эффективно реализовать базирующиеся на системных средствах «вычислительные услуги», но и существенно повысить тематическую квалификацию вычислительной машины за счет согласованного профессионального решения возникающих в ходе разработки прикладных и системных проблем.

Одной из основных форм специализированного программного обеспечения являются пакеты прикладных программ.

Пакет прикладных программ – это комплекс взаимосвязанных прикладных программ и средств системного обеспе-

чения (программных и языковых), предназначенный для автоматизации решения определенного класса задач [12].

Будем в дальнейшем весь круг работ, связанных с разработкой алгоритмов и программ решения задач, а также с подготовкой и проведением расчетов, называть автоматизацией *прикладной деятельности*. Отметим, что всякая конкретная автоматизируемая прикладная деятельность характеризуется двумя факторами. Во-первых, *предметной областью*, т.е. совокупностью решаемых прикладных задач и используемых при этом численных методов, и, во-вторых, *дисциплиной работы*, т.е. системой правил, соглашений, технологических подходов и приемов, принятых при разработке, отладке и эксплуатации программ, в том числе и при проведении расчетов.

В структуре пакета прикладных программ можно выделить три основных компонента: функциональное наполнение, язык заданий и системное наполнение (рис.1) [13].

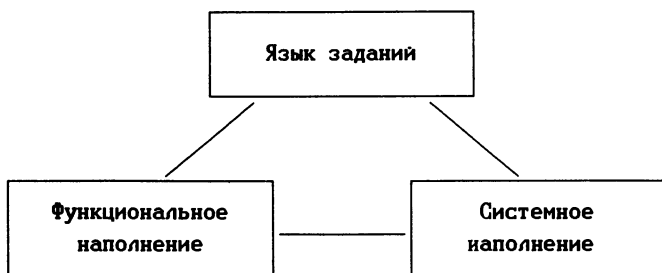


Рис.1. Структура пакета прикладных программ

Функциональное наполнение отражает специфику предметной области пакета и представляет собой совокупность модулей. Под *модулем* здесь понимается конструктивный элемент, используемый на различных стадиях функционирования пакета. Формальное определение модуля, устанавливающее допустимые типы модулей, формы их представления, отношения между модулями, способы хранения и использования модулей, а также формы их сопряжения, будет даваться только в применении к конкретному пакету, на основе учета специфики его языка заданий и системного наполнения. Язык (языки), на котором записываются модули функционального наполнения, будем называть *базовым языком*

пакета. Состав функционального наполнения пакета, его мощность, или полнота охвата («покрытия») им предметной области отражает объем прикладных знаний, заложенных в пакет, т.е. потенциальный уровень тематической квалификации пакета.

Язык заданий пакета является средством общения пользователя с пакетом. Он позволяет описывать последовательность выполнения различных операций, обеспечивающую решение задачи, или постановку задачи, по которой эта последовательность строится автоматически. Кроме того, язык заданий дает возможность формулировать запросы, касающиеся других видов работ, выполняемых в рамках прикладной деятельности, охватываемой пакетом. Допустимый набор операций, лексика и синтаксис языка заданий определяются предметной областью, обслуживаемой пакетом, и реализуемой им дисциплиной работы. Архитектура (т.е. представляющийся пользователю внешний вид) системы определяется тем, какие задачи система может решать и какие возможности она дает пользователю. Ясно, что язык заданий отражает основные архитектурные решения, принятые разработчиками пакета, стремившимися повысить уровень квалификации вычислительной системы в определенной прикладной области. Именно через язык заданий пользователь воспринимает и оценивает, каковы «вычислительные услуги» и насколько удобно их использование, т.е., другими словами, каков фактический уровень тематической квалификации вычислительной системы.

Системное наполнение представляет собой совокупность программ, которые обеспечивают выполнение заданий и взаимодействие пользователя с пакетом, адекватное дисциплине работы в данной прикладной деятельности. Можно говорить, что системное наполнение организует использование потенциала знаний, заложенных в функциональное наполнение, в соответствии с возможностями, предусмотренными в языке заданий пакета. Реализация функций системного наполнения осуществляется на основе согласованного использования а) штатных общецелевых средств системного обеспечения, б) средств системного наполнения, расширяющих и сопрягающих возможности компонентов штатного обеспечения, и в) специальных средств системного наполнения, выполняющих управляющие, архивные и обрабатывающие процедуры с учетом специфики прикладной деятельности, охватываемой пакетом. Язык (языки), на котором пишутся программы системного наполнения, будем называть *инструментальным языком* пакета.

Итак, из проведенного краткого рассмотрения структуры пакета видно, что пакет сочетает в себе знания о вычислениях, проводимых в конкретной прикладной деятельности, с системными средствами, предоставляющими разнообразные формы расширения и использования этих знаний в различных рабочих контекстах.

Эти качества пакета выглядят особенно привлекательно при разработке программного обеспечения, соответствующего тем потребностям в новых формах ведения вычислительных работ, которые прослеживаются в большинстве современных приложений. Если говорить о физико-технических приложениях, где использование вычислительной техники традиционно осуществлялось опережающими темпами, то необходимо прежде всего отметить следующие тенденции [2,9,14]:

- переход к решению комплексных проблем, объединяющих многообразные физические и инженерно-технические направления исследований;
- возрастание роли вычислительных машин при проведении естественно-научных экспериментов;
- расширение круга специалистов, прибегающих к услугам вычислительных систем;
- создание и развитие фондов прикладных программ;
- выдвижение на первый план в программировании вопросов, связанных с эффективной организацией вычислений и конструированием программ на основе модульной структуры математических моделей и алгоритмов;
- появление новых разнообразных форм ведения вычислительных работ как в плане разработки программ, так и в плане проведения расчетов.

Легко видеть, что рассмотренные тенденции выдвигают такие требования к программному обеспечению вычислительных систем, используемых в физико-технических приложениях, которые едва ли могут быть удовлетворены в рамках одного программного проекта-исполния.

Здесь возникает задача создания такого программного обеспечения, которое удовлетворяло бы запросы пользователей, имеющих различные проблемные интересы и обладающих различными навыками в проведении вычислений. Другими словами, речь идет о задаче создания разносторонне квалифицированного программного обеспечения, а эта задача, как отмечалось выше, может быть эффективно решена путем разработки множества пакетов прикладных программ, ориентированных на конкретные разделы работ в приложениях.

Ориентированность каждого пакета на индивидуальные потребности автоматизируемой прикладной деятельности объясняет исключительное многообразие решений как с точки зрения архитектуры, так и с точки зрения структуры пакета. Можно кратко проиллюстрировать это обстоятельство двумя примерами, относящимися к физико-техническим приложениям.

Так, для обслуживания работ, связанных с решением комплексных проблем, необходимо разрабатывать пакеты, обладающие мощным функциональным наполнением с большим запасом в охвате предметной области. При этом часто существует возможность создавать функциональное наполнение на базе уже имеющихся фондов прикладных программ. Язык заданий и системное наполнение таких пакетов в первую очередь должны обеспечивать удобство и эффективность при разработке и конструировании программных комплексов из модулей функционального наполнения, возможность развития функционального наполнения и поддержки различных форм взаимодействия модулей при исполнении программного комплекса, обеспечивать организацию длительных и оперирующих большими объемами данных вычислений. При разработке пакетов, ориентированных на обслуживание комплексных проблем, акцент делается на вопросах технологии построения программных комплексов и организации регулярных расчетов [1,9,14,15]. Кроме того, необходимо учитывать, что основными пользователями таких пакетов являются прикладные программисты, т.е. специалисты, создающие и использующие программный продукт.

Существенно иными качествами должны обладать пакеты, ориентированные на обслуживание естественно-научных экспериментов. Использование современных методов обработки научных наблюдений [16-21] дает возможность значительно расширить спектр функций вычислительной машины в цикле натурального эксперимента от автоматизации процесса управления установкой и первичной обработки результатов эксперимента до математической интерпретации экспериментальных данных, позволяющей получить значения таких физических характеристик, которые нельзя было непосредственно измерить в силу экстремальности условий прохождения эксперимента. В связи с этим большое практическое значение приобретают специализированные программные системы или пакеты программ, обеспечивающие «квалифицированное участие» машины в проведении всех этапов физического эксперимента. Состав функционального наполнения таких пакетов относительно консервативен, так как он определя-

ется ограниченным набором задач, отражающих специфику эксперимента. Часто предъявляются жесткие требования к эффективности программ, в особенности при использовании режима реального времени. Поскольку пользователями пакета являются обычно специалисты (экспериментаторы, проектанты), не имеющие профессиональных навыков в области вычислительной техники, особое внимание следует уделить обеспечению естественных для них форм общения с вычислительной машиной и с экспериментальной установкой в целом.

Хотя рассмотренные нами примеры предназначались лишь для краткой иллюстрации разнообразия подходов при разработке пакетов, однако они отражают и тот факт, что создание пакета является комплексной задачей, для решения которой необходимо совместное участие специалистов из данной прикладной области и системных программистов. Именно тогда, когда это требование было осознано и получило практическую поддержку, начался новый этап в решении задачи повышения уровня тематической квалификации вычислительных систем.

Можно считать (и это подтверждается датами публикаций, посвященных пакетам), что пакетная проблематика как самостоятельная область в программировании, имеющая свои собственные задачи и интересы, сложилась в течение последних полутора десятилетий. За этот период пакет программ эволюционировал от простой тематической подборки программ или большого программного комплекса с рудиментарной системной поддержкой до развитой программной системы, обеспечивающей достаточно полную автоматизацию конкретной прикладной вычислительной деятельности [6,8,22].

В проводимом ниже кратком обзоре мы попытаемся выделить характерные моменты в эволюции пакетов программ, а также отметить те подходы к разработке пакетов, которые отражают текущее положение в этой проблематике. Рассмотрение будем вести в трех аспектах, соответствующих основным структурным компонентам пакета: функциональному наполнению, языку заданий и системному наполнению.

Предшественниками функционального наполнения развитого пакета являются библиотеки стандартных подпрограмм, реализующих элементарные вычислительные процедуры, и простые тематические наборы программ для решения типовых, не связанных между собой задач. (Заметим, что именно такие наборы и породили термин «пакет».) Подобные программные коллекции обычно разрабатываются ведущими

вычислительными организациями и поставляются в комплекте штатного программного обеспечения фирмами-изготовителями вычислительных машин [23-29].

Библиотеки подпрограмм и простые пакеты используются и как вспомогательные средства при создании программ решения больших прикладных задач, и как программный материал, включаемый в состав функционального наполнения пакета. Однако существует качественное различие между библиотекой и простым пакетом с одной стороны и функциональным наполнением пакета с другой.

Дело в том, что разработка библиотеки и простого пакета ведется вне рассмотрения всего объема конкретной прикладной деятельности и обычно преследует цель обеспечить общими «вычислительными услугами» ряд смежных предметных областей. Как правило, специфические задачи каждой из таких смежных областей не находят своего отражения в библиотеке и простом пакете.

Напротив, при разработке функционального наполнения развитого пакета преследуется цель достижения полного охвата конкретной предметной области. Другими словами, состав программного материала функционального наполнения должен обеспечивать решение любой прикладной задачи из числа рассматриваемых в прикладной деятельности на данном этапе развития пакета. В связи с этим требованием разработка функционального наполнения пакета начинается с проведения модульного анализа предметной области [14,30-33].

Целью модульного анализа является построение такого базиса простых алгоритмов, суперпозицией которых могут получаться алгоритмы решения всех задач, рассматриваемых в данной предметной области [9]. С точки зрения функциональной нагрузки элементы алгоритмического базиса могут быть как *математическими*, т.е. описывающими реализации отдельных математических методов, так и *физическими* (или *прикладными*), т.е. соответствующими конкретным физическим задачам или подзадачам [34].

После модульного анализа переходят к программной реализации функционального наполнения. На этом этапе разрабатываются модули – конструктивные элементы, используемые на различных стадиях работы пакета. Подготовка модулей проводится в соответствии с некоторым регламентом модуляризации материала, включаемого в состав функционального наполнения пакета. Такой регламент, разрабатываемый совместно прикладными и системными программистами, устанавливает базовый (напомним, что базовым назы-

вается язык модулей функционального наполнения) язык (языки) программирования пакета, а также допустимые формы представления и взаимодействия модулей. Модулями пакета становятся прежде всего программные реализации тех базисных алгоритмов, которые были выделены на этапе модульного анализа. Для таких модулей регламент модуляризации может, например, потребовать, чтобы все они допускали автономную трансляцию или были представлены в форме процедур базового языка (языков), или же может санкционировать использование макросредств и т.д. Могут регламентироваться и способы установления информационных связей: через параметры, через общие переменные, с помощью специальных интерфейсных средств и т.д. На разрабатываемый регламент модуляризации помимо модульного анализа влияет и планируемая организация вычислительных работ, т.е. дисциплина прикладной деятельности. Этим объясняется появление таких модулей, как совокупность начальных данных, схема счета некоторых типичных задач, фрагмент документации и т.д.

Программная реализация функционального наполнения может осуществляться как путем оригинальной разработки модулей, так и путем адаптации программного материала из имеющихся источников. Адаптацию материала из развитых библиотек и крупных программных фондов облегчает его организация на базе концепций модульного программирования [35-39].

Языки заданий первых специализированных систем программирования и пакетов программ практически совпадали с их базовыми языками программирования, такими, как Фортран, Алгол, ассемблер, или языками управления заданиями операционных и мониторных систем, в терминах которых описывались работы, выполняемые на машине [18,40]. Проблемная ориентация таких языков достигалась в основном путем использования соответствующей мнемоники при идентификации переменных, функций и процедур. Понятно, что такой подход не давал особых преимуществ при составлении задания для пакета по сравнению с обычным программированием, но вместе с тем он и не требовал сколько-либо значительной специализированной системной поддержки. Поэтому неудивительно, что разработчиками и пользователями первых пакетов были прикладные программисты, которые своими усилиями компенсировали недостаточную приспособленность языка заданий пакета к специфике конкретного приложения.

Привлечение системных программистов к разработке пакетов позволило кардинально развить подход к созданию языка заданий пакета на основе одного из штатных языков программирования. Теперь проблемная ориентация языка заданий выражалась не только используемой мнемоникой, но главным образом специальными языковыми макроконструкциями, позволяющими компактно и наглядно формулировать основные фазы процессов решения задач из конкретной предметной области. Такого рода идеология разработки языков заданий, называемых иногда *встроенными* или *вложенными*, имеет и теперь очень много сторонников и получила широкое распространение в практике построения так называемых модульных систем различной проблемной ориентации. Причиной такой популярности встроенных языков, хотя они и требуют от пользователя хороших навыков в программировании, по-видимому, является тот разумный компромисс, который легко достигается между выразительной силой расширяющих язык макроконструкций и стоимостью их системной реализации [41,42].

Следующий этап развития языков заданий пакетов характеризуется появлением *самостоятельных* (т.е. не связанных ни с какими языками программирования) проблемно-ориентированных или *специализированных* языков заданий [17,43,44]. Пользователь такого языка не обязательно является профессиональным программистом. Как правило, проблемно-ориентированный язык заданий основывается на лексике предметной области пакета и имеет очень простой синтаксис. Кроме того, учитывая характер работы и уровень программистской подготовки пользователя, разработчики пакета стремятся обеспечить высокий уровень неprocedureности проблемно-ориентированного языка. Это означает, что большая часть задания пакету, составленного на проблемно-ориентированном языке, является описанием постановки решаемой задачи (т.е. указывает, «что делать»), а меньшая – описанием процесса решения этой задачи (т.е. определяет, «как делать»).

Концепция проблемно-ориентированных языков, основанных на аппарате специализированных программирующих программ, была предложена В.М.Глушковым еще в 1959г. [45]. Однако только в рамках работ по пакетной проблематике она получила свое практическое воплощение. Это объясняется тем, что большая семантическая насыщенность конструкций проблемно-ориентированного языка требует развитых программных средств для их реализации. В пакете указанная трудность преодолевается за счет удачного структур-

ного решения: основная нагрузка ложится на модули функционального наполнения, программно реализующие знания о вычислениях, проводимых в предметной области, а интерпретация задания сводится к вызову соответствующих модулей и организации информационных связей между ними.

В настоящее время получило распространение создание пакетов с помощью специально разработанных для этой цели инструментально-базовых систем, т.е. систем, содержащих инструментальные средства для создания будущих пакетов, а также базовые заготовки для включения в состав этих пакетов. При этом применяются два подхода к разработке языка заданий пакета [6].

Первый основан на идее использования языка-ядра, являющегося общей частью языков заданий всех пакетов, создаваемых с помощью инструментально-базовой системы. В качестве языка-ядра обычно служит входной язык заданий самой системы. Этот язык, помимо средств, отражающих все то общее, что есть в приложениях, для которых будут создаваться пакеты, содержит также средства расширения. При рассматриваемом подходе язык заданий конкретного пакета создается путем дополнения языка-ядра новыми понятиями и конструкциями, которые определяются с помощью средств расширения и учитывают специфику приложения [46-48].

Другой подход предполагает, что входной язык инструментально-базовой системы является метаязыком (т.е. языком для описания языков), посредством которого определяются синтаксис и семантика языка заданий каждого создаваемого пакета. Этот подход в случае достаточно развитого метаязыка обеспечивает значительную независимость инструментально-базовой системы от приложений и позволяет определять совершенно различные по форме языки заданий пакетов [49,50].

Средства системного наполнения развитого пакета берут свое начало от первых компилирующих и интерпретирующих систем, которые автоматизировали процессы сборки основной программы пользователя из отдельных частей и включения в нее библиотечных подпрограмм [51-54].

В первых специализированных системах программирования и пакетах программ (например, [18,55]), языки заданий которых совпадали с базовыми языками программирования, для этих целей использовались либо отдельные средства штатного программного обеспечения, такие как компиляторы, макрогенераторы, редакторы внешних связей, редакторы

текстов, либо основанные на них многофункциональные системы автоматизации программирования [56,57]. Однако уже в первых пакетах функции системного наполнения не ограничивались только организацией взаимодействия пакета с перечисленными выше компонентами штатного программного обеспечения, а распространялись также и на специфические процессы в работе пакета. Например, в специализированной системе программирования HYDRA с помощью системных подпрограмм решались вопросы динамического распределения оперативной памяти, формирования специфичных структур данных и доступа к ним, а также организации взаимодействия модулей.

В дальнейшем, с появлением большего разнообразия форм охвата предметной области, получили распространение языки заданий, включающие в себя средства поддержки всего диапазона действий, выполняемых проблемным программистом при решении конкретных задач. Оказалось, что специализация языка позволяет существенно повысить производительность труда проблемного программиста на любом этапе его работы. Реализация такого подхода потребовала подключения широкого набора средств системного программирования, состав которых характерен и для других современных форм пакетов [15,34,58-61].

Действительно, знакомство с внутренними структурами этих пакетов позволяет выделить такие ставшие уже традиционными составляющие системного наполнения (рис.2):

- резидентный монитор, осуществляющий интерфейс как между отдельными компонентами системного наполнения, так и между ними и штатным программным обеспечением;
- транслятор входных заданий, формирующий внутреннее представление задания и реализуемый обычно в виде макрогенератора или препроцессора;
- интерпретатор внутреннего представления задания;
- архив функционального наполнения (подсистема хранения программного материала);
- банк расчетных данных (подсистема хранения начальных и промежуточных данных, а также результатов расчетов);
- монитор организации вычислительного процесса (взаимодействие модулей по данным и управлению);
- планировщик вычислительного процесса, который определяет последовательность выполнения модулей, реализующую задание пакету;
- монитор организации интерактивного взаимодействия с пользователем.

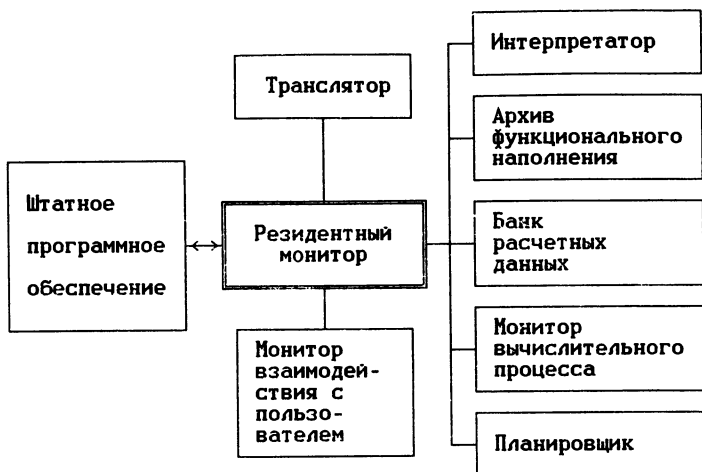


Рис 2. Компоненты системного наполнения пакета

В различных пакетах перечисленные компоненты получили разную степень развития, иногда просто отсутствовали, реализовывались на основе различных системных подходов и по-разному назывались (терминология в пакетной проблематике, к сожалению, не вполне установилась). Поэтому приведенную выше структуру следует воспринимать скорее как сводный список различных известных компонентов системного наполнения, а не как образ конкретного или типичного пакета.

Другое современное направление развития пакетов – обеспечение удобства работы специалистов, не знакомых с программированием. Акцент тут делается на достижении высокой степени непроецедурности языка заданий, скрывающего от пользователя основную массу алгоритмических подробностей решения его задачи. Среди компонентов системного наполнения здесь на первый план выходят транслятор входного задания, планировщик вычислений и монитор организации интерактивного взаимодействия с пользователем. Кроме того, в таких пакетах развиты средства информационно-справочного обслуживания [44,46].

Мы уже отмечали, что одной из важных тенденций в пакетной проблематике является переход к построению пакетов с помощью инструментально-базовых систем. Посколь-

ку часто требуется, чтобы при формировании функционального наполнения пакета можно было использовать уже имеющиеся прикладные программы и библиотеки подпрограмм, в инструментально-базовых системах обычно предусматриваются средства поддержки подключения таких программных материалов. Во всех других аспектах программная часть инструментально-базовой системы реализуется так, чтобы ее можно было использовать в качестве базы системного наполнения создаваемого пакета.

Заканчивая краткий обзор развития пакетной проблематики, мы хотим обратить внимание на то, что эволюция пакетов протекает в пространстве двух измерений: прикладного и системного и всякий односторонний подход к созданию пакета не может принести успеха. Действительно, и библиотека подпрограмм, и тематический набор прикладных программ, полностью или даже с запасом охватывающие некоторую предметную область, сопоставимы всего лишь с функциональным наполнением пакета и возлагают на пользователя всю работу по организации совместного выполнения подпрограмм и программ в различных комбинациях при решении каждой конкретной задачи. С другой стороны, любой универсальный язык программирования и его реализация, выполненные вне анализа конкретной прикладной деятельности, не смогут служить адекватным средством автоматизации вычислений в данной области. Во-первых, в таком языке не найдут свое отражение привычные для специалистов данной прикладной деятельности специфические терминология и технологические приемы ведения вычислений. Во-вторых, такой подход требует, вообще говоря, чтобы для решения каждой задачи разрабатывались соответствующие вычислительные процедуры. Особенно существенно, что оба этих односторонних подхода требуют от пользователя высокой профессиональной программистской подготовки.

Пакетная проблематика переживает период экстенсивного развития. Увеличивается как число вычислительных организаций, в деятельности которых пакетная тематика занимает одно из ведущих мест, так и круг приложений, для которых разработаны, создаются или проектируются пакеты программ. Одновременно с этим происходит и интенсивное развитие пакетов, выражающееся прежде всего в создании более эффективных и надежных алгоритмов и прикладных программ, а также в разработке разнообразных методов и форм организации системного наполнения.

В этой книге систематизируются задачи, стоящие перед пакетами прикладных программ, и исследуются пути их решения. Изложение базируется на рассмотрении ряда конкретных производственных пакетов. Предметными областями для описываемых пакетов являются, как правило, различные классы задач вычислительной физики, однако использованные в них системные решения могут быть применены при других разнообразных предметных ориентациях пакетов.

При изложении материала особое внимание уделяется следующим вопросам:

- классификации языковых и программных средств системного обеспечения пакетов программ и выявлению основных функций пакета, определяющих архитектуру (внешние функциональные возможности) его системного наполнения;
- анализу влияния прикладной деятельности на архитектуру пакета в целом и связей между конкретными характеристиками прикладной деятельности и формами реализации той или иной системной функции пакета;
- исследованию (главным образом в сфере приложений вычислительной физики) возможности эффективной реализации различных комбинаций выделенных системных функций пакетов и практической реализации производственных пакетов программ.

Книга состоит из шести глав.

В гл.1 рассматривается архитектура системного обеспечения пакетов прикладных программ. Вводятся основные понятия, используемые при разработке системного обеспечения: язык заданий, регламент модуляризации функционального наполнения, планирование вычислений, операционные возможности, информационное обслуживание. Анализируются характерные формы реализации этих понятий в пакетах, решающих инженерно-физические задачи.

Последующие главы посвящены описанию ряда производственных пакетов, разработанных в ИПМ имени М.В.Келдыша АН СССР для различных классов задач математической физики. Главная особенность рассматриваемых пакетов – многообразие решаемых ими задач, повлекшее за собой применение широкого спектра различных системных решений. Среди этих решений нашли свое отражение многочисленные формы реализации каждого из основных понятий, выделенных в гл.1. Единственная существенная функция, не реализованная в производственных пакетах ИПМ, – планирование вычислений, для исследования которого потребовалось специально разработать методический пакет, описываемый в конце гл.6.

Авторы надеются, что рассмотренные в книге методы и средства окажутся полезными при выборе архитектуры и реализации системного обеспечения новых пакетов прикладных программ.

*

*

*

Авторы выражают признательность В.Я.Карпову, одному из инициаторов и постоянных участников проведенного цикла работ по пакетам программ. Авторы благодарят всех своих коллег – разработчиков и пользователей конкретных пакетов – и в первую очередь А.И.Горлина, В.И.Кротова, Э.С.Луховицкую и М.П.Матекина, ознакомившихся с содержанием отдельных разделов книги и сделавших ряд важных замечаний.

Г л а в а 1

АРХИТЕКТУРА ПАКЕТА ПРИКЛАДНЫХ ПРОГРАММ

1.1. Системные характеристики пакета

Как отмечалось во введении, пакеты прикладных программ служат средством повышения эффективности общения человека с вычислительной машиной при решении прикладных задач. В основе успеха разработки каждого конкретного пакета лежит всесторонний учет специфики автоматизируемой прикладной деятельности.

Многообразие приложений (предметных областей и дисциплин работ) закономерно влечет за собой и многообразие пакетов. Следует однако различать две составляющие такого многообразия, связанные с *функциональным* и *системным* наполнениями пакета.

Многообразие функциональных наполнений пакетов вытекает из природы обслуживаемых ими предметных областей. Здесь попытка систематизации (если даже ограничиться областью физико-технических приложений) вывела бы нас далеко за рамки задач настоящей книги.

Напротив, многообразие системных методов и средств, применяемых при построении пакетов, на сегодняшний день не столь велико. В настоящей главе анализируются основные системные проблемы, с которыми сталкиваются разработчики пакетов. Вводимые в ней понятия и терминология позволяют оценить с единых позиций системные решения, принятые в ряде описываемых в последующих главах конкретных производственных пакетов.

К числу наиболее важных понятий, реализация которых в пакете определяет его системные характеристики, относятся:

- язык заданий (средство общения пользователя с пакетом);
- регламент модуляризации функционального наполнения (методы декомпозиции программного материала, правила оформления включаемых в пакет прикладных программ);

- планирование вычислений (способ построения расчетной цепочки, т.е. последовательности функциональных и системных модулей, обеспечивающей выполнение задания пользователя);

- операционные возможности (способ исполнения расчетной цепочки, интерфейс со штатным программным обеспечением, организация межмодульного интерфейса, управление вычислительным процессом, хранение программного материала и расчетных данных, интерактивное взаимодействие пользователя с пакетом, системные услуги);

- информационное обслуживание (удовлетворение информационных запросов, поступающих от пользователей или от программных компонентов пакета).

В последующих разделах рассматриваются различные подходы к реализации этих понятий в пакетах, ориентированных на физико-технические приложения.

1.2. Язык заданий

Общая структура и стиль языка заданий пакета в значительной степени зависят от дисциплины работы, принятой в обслуживаемой пакетом предметной области. Можно выделить две основные (в определенном смысле противоположные) дисциплины проведения вычислений:

- *активную дисциплину*, предусматривающую при создании конкретных расчетных программ модификацию и настройку имеющихся модулей функционального наполнения, а также разработку новых модулей;

- *пассивную дисциплину*, предусматривающую проведение вычислений без модификации функционального наполнения пакета.

Активная дисциплина работы свойственна деятельности прикладных математиков, владеющих методами решения задач из соответствующей предметной области и создающих программное обеспечение для решения этих задач.

Пассивная дисциплина характерна для деятельности так называемых *конечных* пользователей, т.е. специалистов, которые не обязательно имеют высокий уровень подготовки в области вычислительной техники и используют машину в качестве высококвалифицированного вычислителя, обладающего навыками решения специфических прикладных задач.

Такое выделение двух дисциплин работы достаточно условно (можно считать, что они соответствуют краям спектра реально существующих дисциплин) и преследует цель подчеркнуть контрастность системных подходов, ис-

пользуемых при автоматизации приложений с различными стилями ведения вычислительных работ.

Так, характерной особенностью языков заданий пакетов, обслуживающих проведение вычислительных работ в режиме активной дисциплины, является их направленность на описание схем программ решения прикладных задач. Причем центральное место в таких языках (мы будем называть их *языками сборки*) занимают не средства описания данных и манипулирования ими, что свойственно универсальным процедурно-ориентированным языкам программирования, а средства:

- конструирования схем программ, в которых указывается порядок выполнения и взаимодействия модулей при решении определенной задачи;

- развития или модификации функционального наполнения;

- управления процессами генерации и исполнения расчетной программы, реализующей задание пользователя.

Языки сборки, предоставляющие возможность применения столь разноплановых средств в рамках одного задания, как правило, требуют, чтобы пользователи имели достаточную профессиональную программистскую подготовку. Однако, если учесть, что такими пользователями являются прикладные программисты, то эти требования представляются вполне приемлемыми [18,56,62-66].

Отметим, что уровень требований, предъявляемых к прикладным программистам, решающим задачи с помощью пакета, обычно все же несколько ниже уровня требований, которые предъявлялись бы к ним при проведении тех же самых работ средствами штатного программного обеспечения. Однако для пакетов с активной дисциплиной проведения расчетов на первый план нередко выдвигается не столько снижение профессиональных требований к пользователям, сколько повышение производительности их труда. Повышение производительности достигается здесь, в частности, за счет использования специализированных системных средств модификации и сборки программ, отсутствующих в среде универсального штатного программного обеспечения. Эти различия в уровнях требований и в производительности труда характеризуют то, насколько повышается тематическая квалификация вычислительной системы в случае использования данного пакета программ при решении рассматриваемого класса задач.

Главная цель разработчика языка заданий пакета, обеспечивающего решение прикладных задач в режиме пассивной

дисциплины, заключена в том, чтобы «спрятать» от конечного пользователя основную массу алгоритмических подробностей решения его задачи, или, другими словами, повысить уровень непроцедурности языка [67]. Такие языки, называемые *языками запросов*, ориентированы обычно на формулирование содержательных постановок задач, т.е. запросов, указывающих, «что необходимо получить» без явного задания того, «как это получить» [48]. Пользователь тем самым избавляется от необходимости конкретизировать способы и средства решения его задачи, что позволяет радикально понизить порог требований к уровню его программистской подготовки.

Повышение тематической квалификации вычислительной системы определяется здесь тем множеством прикладных задач, которые можно формулировать и решать с помощью языка запросов. По мере развития функционального наполнения пакета может расти и круг решаемых им задач, и поэтому язык запросов обычно определяется как расширяемый, т.е. допускающий включение новых запросов. Расширяемость языка может поддерживаться соответствующей инструментально-базовой системой [49,50] или же достигается путем прямого расширения возможностей системного наполнения.

Язык заданий пакета может быть реализован как в форме самостоятельного языка, так и в форме встроенного языка, т.е. расширения существующего языка программирования. Независимо от формы реализации разработчик языка стремится к тому, чтобы лексика, синтаксис и семантика языковых конструкций были бы как можно ближе к пользовательскому восприятию решаемых прикладных задач. Важно не только опираться на понятия и терминологию предметной области и свойственной ей дисциплины работы, но и наглядно отразить в языке основные рабочие акты автоматизируемой прикладной деятельности.

1.3. Пакетный подход к модуляризации программ

Одной из ключевых проблем разработки пакета прикладных программ является *модуляризация*, т.е. разбиение функционального наполнения пакета на модули. Тщательно выполненный модульный анализ предметной области и проведенная на его основе модуляризация позволяют сократить объем работ по реализации пакета, повышают его надежность и облегчают последующее развитие.

Напомним, что под *модулем* мы понимаем конструктивный элемент, используемый на различных стадиях функционирования пакета. Число разнообразных форм модулей, используемых в пакетах, весьма велико. Прежде всего следует выделить программные модули, модули данных и модули документации. Для программных модулей известны, например, такие (часто переходящие одна в другую) формы, как: подпрограмма; конструкция алгоритмического языка, допускающая автономную трансляцию; макроопределение; файл, содержащий такой текст фрагмента программы, который рассматривается как самостоятельный объект для изучения и/или редактирования; набор указаний, задающих способ построения конкретной версии программы; реализация абстрактного типа данных и др.

Те или иные формы модулей, а также совокупность применимых к модулям операций выбираются в каждом пакете исходя из специфики автоматизируемой им прикладной деятельности. Для того, чтобы четче определить особенности модуляризации в пакетах программ, рассмотрим сначала некоторые мотивы выделения модулей, имеющие более широкое применение, т.е. используемые не только в пакетах, но и в других программных разработках.

Наиболее простой и очевидный из этих мотивов — *экономия записи*. Повторяющиеся или близкие по содержанию части программы оформляются в виде отдельного модуля (подпрограммы, макроопределения и т.п.). Текст программы сокращается, поскольку на месте выделенных частей записываются лишь обращения к выделенному модулю.

Сокращение суммарного времени трансляции достигается за счет выделения модулей, допускающих автономную трансляцию. Тогда при внесении изменений достаточно будет перетранслировать только затронутые ими модули, а не всю программу.

Удобство редактирования текста программы требует разбиения больших программ на более мелкие части, которые выступают в качестве обозримых и потому более удобных объектов редактирования.

Соображения *наглядности*, или легкости восприятия программы также ведут к разбиению ее на части. Тут выделяются модули, логику которых можно «охватить одним взглядом» или, скажем, целиком показать на странице листинга или на экране дисплея.

При *распределении работ* по написанию программы между несколькими исполнителями в качестве модулей выступают части, реализуемые отдельными исполнителями. Обычно

стремятся к минимизации связей между такими частями, что позволяет сократить количество и интенсивность совместных обсуждений, проводимых в ходе разработки.

Потребность *размещения длинной программы в оперативной памяти ограниченного объема* нередко приводит к выделению модулей (сегментов), сменяющих друг друга в оперативной памяти во время выполнения программы. Основным критерий успеха модуляризации при сегментировании — минимизация требуемого числа подкачек сегментов из внешней памяти.

Перечисленные выше мотивы модуляризации связаны с отдельными сторонами ведения программного хозяйства и поэтому имеют технический характер. В любом крупном программном проекте приходится в той или иной степени сталкиваться с каждым из этих мотивов. Однако при анализе причин декомпозиции программ на модули на первый план обычно выдвигаются соображения *систематизации и упрощения разработки* сложных программ. Исходя из этих соображений, рекомендуется выделять в качестве модулей функционально самостоятельные независимые части программы.

Как отмечает Дейкстра [68], в основе такой модуляризации лежит определенная дисциплина мышления. Согласно этой дисциплине, решение сложной проблемы можно упростить, разложив ее на ряд относительно независимых аспектов, каждый из которых допускает изолированное изучение. Упомянутое разложение естественным образом отражается в структуре обслуживающей данную проблему программы: каждому из выделенных аспектов соответствует отдельный программный модуль.

Технические мотивы модуляризации, как правило, неплохо согласуются друг с другом. Дело в том, что для большинства из них допустим известный произвол в выборе конкретных границ модуля. Примирить требования различных мотивов можно, если в пределах этого произвола выбирать границы, руководствуясь соображениями функциональной самостоятельности и независимости модулей.

В частности, при этом улучшается наглядность, поскольку сокращается число обрабатываемых модулем объектов и, главное, становится возможным достаточно четко сформулировать назначение модуля. Распределение работ между исполнителями, проведенное на основе функциональной модуляризации, приводит к выделению относительно независимых модулей и, следовательно, минимизирует число связей между ними. Наконец, удобство редактирования и экономия

времени трансляции достигаются за счет того, что изменения программы, касающиеся определенного понятия, как правило, локализуются в одном модуле и поэтому только один модуль приходится редактировать и перетранслировать при внесении этих изменений.

Тут мы подошли к еще одному важному аргументу в пользу функциональной модуляризации. С ее помощью удается облегчить развитие программы, так как она, вообще говоря, обеспечивает локализацию вносимых в программу изменений. Парнас [69] предлагает оформлять в виде модуля любое решение, принимаемое в ходе разработки программы, «скрывая» тем самым его последствия от остальных частей программы и облегчая возможность последующего пересмотра этого решения. Практически все большие программы (в том числе и пакеты прикладных программ) претерпевают за время своего существования многочисленные изменения, и облегчение выполнения таких изменений является весьма актуальной задачей.

При конструировании пакетов прикладных программ функциональной модуляризации придается новое содержание. Здесь не просто выделяют функционально самостоятельные части, а пытаются сформировать такой их набор, который «покрывает» обслуживаемую предметную область. Покрытие области означает, что для любой ставящейся в ней задачи может быть построена решающая ее расчетная программа, представляющая собой необходимым образом организованное подмножество модулей из выделенного набора.

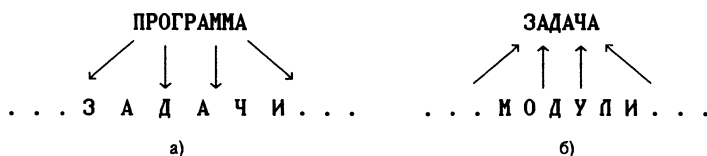


Рис 11 Два подхода к модуляризации:
универсальная программа (а); пакет (б) [63]

На рис.1.1 проиллюстрировано отличие пакетного подхода к модуляризации от разработки обычных «универсальных» программ, где с помощью одной программы стараются охватить возможно более широкий класс задач. На первый взгляд отличие заключается в том, что в обоих случаях создаются одни и те же части программы, но в выполняемую универсальную программу включается весь накопленный программный материал, а в расчетную программу пакета – лишь

некоторые отобранные модули. Конечно же, суть дела не в этом. Если бы требовалось всего лишь сократить размер выполняемой программы, этого можно было бы достичь, например, отсекая ненужные в конкретном расчете куски посредством техники смешанных вычислений, т.е. «специализируя» («конкретизируя») универсальную программу постановкой решаемой задачи.

Главное достоинство пакетного подхода к модуляризации – возможность *безболезненного развития* программного фонда. Расширение класса решаемых пакетом задач может достигаться в основном за счет подключения к пакету дополнительных вновь создаваемых модулей; при этом, вообще говоря, не требуется изменения модулей, существовавших ранее. Напротив, развитие «универсальной» программы всегда сопряжено с изменениями текстов написанных ранее частей программы, что может повлечь за собой отказы при счете отлаженных до внесения изменения вариантов. Как уже отмечалось, простота выполнения изменений является одним из важнейших качеств большой программы, и именно поэтому пакетный подход к модуляризации получил широкое распространение.

Вместе с тем необходимо отметить, что с пакетным подходом связана серьезная технологическая проблема – как организовать взаимодействие существующих модулей с вновь создаваемыми?

Эта проблема легко решается в смежной с пакетами области – в библиотеках программ. Специфика библиотек заключается в том, что при их создании, в отличие от пакетов, не ставится цель покрытия предметной области, т.е. формирования расчетных программ исключительно из хранящихся в библиотеке модулей. Обычно в расчете библиотечные модули соседствуют с модулями, разрабатываемыми и хранимыми вне библиотеки, за счет которых и организуется взаимодействие. Таким образом, требования к библиотечным модулям несколько иные, чем к модулям пакета. Разработчик библиотеки «коллекционирует» модули, заботясь, в основном, лишь о единообразном описании их возможностей. Механическое пополнение коллекции библиотечных модулей – пример редко встречающейся в программировании ситуации, когда без специальных усилий со стороны разработчика развитие программного фонда идет безболезненно, не затрагивая накопленных ранее программ.

В пакетах расчетная программа целиком составляется из модулей пакета (что позволяет, в частности, использовать пакет людям, не знакомым с программированием). Поэтому

для осуществления безболезненного подключения к пакету новых модулей требуются определенные организационные усилия.

Известно решение проблемы организации взаимодействия модулей пакета в рамках технологии искусственного интеллекта. С каждым модулем связывается и записывается в функциональное наполнение формальное описание его входных/выходных данных и условий его применимости, а вместо непосредственных вызовов модулей используются вызовы «по образцу», определяющие не конкретный модуль, а лишь стоящую перед модулем задачу [48,70,71]. Поэтому вновь появившийся модуль никак не воздействует на тексты модулей – своих потенциальных потребителей. Эти модули автоматически найдут его из-за соответствия его описания выставленным ими запросам.

В существующих пакетах программ технология искусственного интеллекта применяется относительно редко, так как она часто приводит к значительной потере эффективности и требует от разработчиков пакета свободного владения весьма нетривиальным логическим аппаратом. В большинстве случаев удается найти более экономичное решение стоящих перед пакетом задач на основе существенно более простых программных конструкций, обеспечивающих, тем не менее, свойство безболезненного выполнения изменений.

Такие конструкции будут рассмотрены в разделе 1.5, посвященном конфигурациям модулей в пакетах, и в последующих главах, описывающих конкретные производственные пакеты.

1.4. Регламент модуляризации функционального наполнения

Вернемся к определению модуля как конструктивного элемента, используемого на различных стадиях функционирования пакета. Что понимается здесь под *конструктивностью* модуля?

Прежде всего имеется в виду *алгоритмическая* конструктивность, которой был в основном посвящен предыдущий раздел. Модуль представляет собой элемент полученного в результате модульного анализа предметной области алгоритмического базиса, служащего основой для построения программ, удовлетворяющих произвольные запросы прикладной деятельности. Кроме того, на алгоритмическую конструктивность модулей влияют структуры типичных вычислительных алгоритмов, связи между элементами алгоритмическо-

го базиса, используемые в этих структурах, информационные потоки, возникающие в различных расчетных контекстах.

Помимо алгоритмической, следует выделить и *технологическую* конструктивность модулей, привносимую дисциплиной работы в приложении и системной средой, на базе которой разрабатывается и эксплуатируется пакет. На технологическую конструктивность воздействуют такие факторы, как:

- формы представления программных модулей;
- виды управляющих связей между отдельными частями программных комплексов (открытые и закрытые подпрограммы, сопрограммы);
- способы организации информационных связей (через аппарат параметров, общие области памяти, общие файлы);
- методы разработки (сверху вниз, снизу вверх и др.) программных комплексов, применяемые в приложении;
- базовый язык или языки программирования, используемые при подготовке прикладных программ;
- ограничения на размеры прикладных программ;
- возможности штатных системных средств, обеспечивающих редактирование связей, загрузку и сегментацию программных комплексов, редактирование текстов.

Требования, вытекающие из алгоритмической и технологической конструктивности, а также из некоторых других рассматриваемых ниже свойств модулей, составляют в совокупности *регламент модуляризации*, т.е. принятую разработчиками пакета форму представления материала в функциональном наполнении, а также способы его создания и развития. Если описание языка заданий рассматривать как спецификацию сопряжения пользователя с пакетом программ [72], то посредством регламента модуляризации определяется сопряжение с пакетом (точнее, с функциональным наполнением пакета) его разработчиков.

Одним из важных свойств модулей, обеспечиваемых регламентом модуляризации, является *совместимость*, т.е. возможность их совместной работы в рамках расчетных программ.

Наиболее остро проблема совместимости модулей встает в том случае, когда к моменту начала работ над некоторым пакетом уже существует значительный программный фонд, созданный независимыми группами разработчиков. Созданный ранее фонд нередко оказывается нестандартизированным. При включении его в функциональное наполнение пакета образуется совокупность подпрограмм, которые функционально полностью покрывают предметную область, но не могут

быть сопряжены друг с другом в рамках одной программы, поскольку они создавались независимо и поэтому не были написаны в расчете на совместную работу. Такие подпрограммы, функционально входящие в алгоритмический базис пакета, но неспособные к непосредственному взаимодействию в расчетных программах, будем называть *квазимодулями*. Препятствием для объединения квазимодулей является обычно отличие в формах представления и хранения совместно используемых данных.

Проблема нестандартизированного фонда допускает два различных решения. Первое, весьма трудоемкое и в силу этого обычно неприемлемое, — заново переписать все квазимодули в соответствии со стандартами, принятыми в пакете. Второе решение — обеспечить такую технологию создания функционального наполнения пакета, которая потребовала бы лишь незначительной доработки программного материала.

Таким образом, можно выделить два подхода к формированию регламента: внутреннюю и внешнюю модуляризацию.

Внутренняя модуляризация предполагает, что совместимость модулей достигается за счет соблюдения всех требований регламента в период разработки программных тел модулей. Регламент фиксирует, в частности, способы организации интерфейса между модулями как по управлению, так и по данным. Внутреннюю модуляризацию используют при разработке пакета «с нуля». Кроме того, если программный фонд уже существует, но стандартизирован в соответствии с некоторым регламентом, регламент модуляризации пакета может наследовать регламент программного фонда, наследуя тем самым и совместимость модулей. При внутренней модуляризации обычно не требуется специальных системных средств поддержки межмодульных интерфейсов, совместимость модулей обеспечивается штатными средствами редактирования межмодульных связей [17,46,65].

При работе с нестандартизированным фондом можно воспользоваться *внешней модуляризацией*. Исходный текст квазимодулей остается неизменным, но каждый квазимодуль дополняется сопроводительной системной записью, называемой паспортом или *заголовком* модуля, в котором содержится формальное описание всех его входных и выходных объектов [47,73–77]. Располагая записанной в заголовке информацией, системное наполнение пакета способно обеспечить весь необходимый межмодульный интерфейс. Таким образом, пара «квазимодуль — заголовок» превращается в полноценный модуль пакета.

Внешняя модуляризация обычно не предъявляет каких-либо специальных требований к оформлению квазимодулей (позволяет рассматривать их как «черные ящики») и тем самым помогает относительно просто адаптировать к требованиям пакета поступающие извне нестандартизированные программные материалы. Плата за простоту адаптации — потребность в значительно более развитых по сравнению с внутренней модуляризацией средствах системного обеспечения. Здесь необходимы язык, на котором описываются заголовки модулей, и системные средства, «понимающие» эти заголовки при организации межмодульного интерфейса.

Разработка языка описания заголовков модулей и системных средств интерпретации этого языка оправдана только в тех случаях, когда адаптируемые квазимодули должны сочетаться в многочисленных разнообразных конфигурациях. Если же планируемое число комбинаций квазимодулей в формируемых расчетных программах невелико и может быть точно определено заранее, внешнюю модуляризацию имеет смысл выполнять путем разработки для каждого квазимодуля специальной программы (программного адаптера), обеспечивающей его совместимость с конкретными модулями функционального наполнения.

В пакетах, осуществляющих планирование вычислений, применяется *смешанная модуляризация*, сочетающая возможности обоих подходов. Тут за счет соблюдения соответствующего внутренней модуляризации регламента программирования функционального наполнения минимизируют издержки, связанные с организацией межмодульного интерфейса, а средства внешней модуляризации используют для описания семантики модулей, условий их применимости, цен и других показателей, необходимых для автоматического планирования вычислений [71].

1.5. Конфигурации модулей

Итак, мы выяснили, каким образом проводится модульный анализ предметной области и как на его основе формируется базисный набор программных модулей пакета. Рассмотрим теперь, какие конфигурации могут образовывать эти модули в расчетных программах и какие существуют способы задания конкретных конфигураций.

Во многих пакетах все расчеты ведутся по одной слабо меняющейся схеме. Ей соответствует вполне определенная структура всех формируемых пакетом расчетных программ. Многообразие проводимых расчетов здесь достигается не

количеством схем счета, а многочисленными вариантами заполнения гнезд фиксированной программной структуры — каркаса.

Например, в функциональном наполнении пакета может содержаться несколько модулей, реализующих различные методы расчета. Но в структуре программы для них предназначается единственное гнездо «МЕТОД», в которое по указанию пользователя при формировании расчетных программ подставляется тот или иной модуль.

При данном подходе, который мы будем называть *каркасным*, не накладывается ограничений на способы взаимодействия модулей. Однако свободой выбора способов взаимодействия разработчик пакета пользуется только на стадии проектирования каркаса программы, а в дальнейшем все модули должны программироваться в строгом соответствии с выработанными при проектировании каркаса требованиями к заполнению гнезд.

Единообразное оформление модулей, принадлежащих одному гнезду каркаса, позволяет достаточно свободно сочетать в вариантах расчетных программ различные комбинации модулей. Можно численно оценить мощность множества расчетных программ, генерируемых пакетом при каркасном подходе. Пусть в спроектированном каркасе содержится n гнезд и для i -го гнезда ($1 \leq i \leq n$) существует m_i различных реализующих его модулей. Тогда число P расчетных программ, которые могут быть, вообще говоря, порождены на базе данного функционального наполнения, выражается формулой

$$P = \prod_{i=1}^n m_i. \quad (1)$$

Отметим, что при реальных расчетах возникают не все потенциально допустимые комбинации модулей. Тем не менее во многих производственных пакетах фактически используемое число вариантов близко к приведенной оценке.

Нетрудно видеть, что каркасный подход обеспечивает безболезненность (см. п.1.3) развития программного фонда пакета. Действительно, развитие фонда идет, как правило, за счет написания новых модулей-реализаций для выделенных гнезд каркаса программы. Появление такого нового модуля никак не затрагивает тексты ни его соседей по гнезду, ни вызывающих его модулей, т.е. проходит безболезненно. Соседи по гнезду никогда не соседствуют в расчетных программах, а вызывающие модули программирова-

лись в рамках соглашений каркаса и потому могут с равным успехом обращаться к любой (в том числе и к новой) реализации гнезда.

При каркасном подходе для задания конкретной конфигурации необходимо для каждого гнезда каркаса программы указать, какой именно модуль из соотнесенного гнезду семейства должен быть помещен в это гнездо. Для этого на языке заданий пакета записывается совокупность пар вида

$$\langle \text{имя гнезда} \rangle = \langle \text{имя модуля} \rangle$$

которая полностью определяет расчетный вариант. Некоторые способы упрощения задания конфигурации при каркасном подходе будут разбираться в последующих главах при рассмотрении производственных пакетов.

Реже встречаются предметные области, в которых многообразие связей модулей по управлению удастся ограничить последовательным выполнением модулей, не жертвуя при этом ни удобством составления программ, ни эффективностью их выполнения. Расчетная программа здесь представляет собой цепочку выполняемых друг за другом модулей, из-за чего данный подход будем называть *цепочечным*.

Обычно состав и порядок следования модулей в цепочке определяется некоторой внешней конструкцией, называемой планом вычислений. Но встречаются и пакеты, где в ходе выполнения цепочки каждый очередной модуль сам динамически выбирает себе преемника.

Каждый из модулей цепочки может вызывать какие-либо подпрограммы. Однако с точки зрения цепочечного подхода внутренние управляющие структуры звеньев цепочки несущественны, т.е. вызываемые модулями подпрограммы обезличены и цепочка воспринимается как линейная одноуровневая структура. Все манипуляции при формировании расчетных вариантов программ не выходят, вообще говоря, за рамки различных последовательностей модулей, вызываемых на одном, самом верхнем уровне. Напомним, что при каркасном подходе модули, формирующие расчетный вариант, заполняют гнезда каркаса программы, которые могут располагаться на любом уровне иерархии управляющих связей.

Столь жесткое ограничение, накладываемое на управляющие связи цепочечным подходом, отчасти компенсируется тем, что положение модуля в цепочке, в отличие от каркасного подхода, не фиксируется. Любой модуль может быть, вообще говоря, включен в любую позицию цепочки.

Если считать, что модули в цепочке могут многократно повторяться, то можно заключить, что потенциально цепоч-

чечный подход на любом непустом наборе модулей позволяет генерировать бесконечное число расчетных программ. Если даже ограничиться не более чем однократным включением модуля в цепочку и считать порядок модулей в цепочке несущественным (в реальных пакетах нередко ни то, ни другое ограничение не выполняется), то тем не менее оценка числа P потенциально возможных расчетных программ составит

$$P = 2^M, \quad (2)$$

где M — число модулей в пакете. Даже такая заниженная оценка выглядит намного внушительнее, чем оценка (1) каркасного подхода. Однако на практике при достаточно большом M подавляющая часть потенциально возможных цепочек никогда не реализуется.

Популярности цепочечного подхода в немалой степени способствует то обстоятельство, что многие современные операционные системы (прежде всего UNIX) предоставляют удобные средства для применения цепочечного подхода при организации связей между задачами. В частности, в языках управления заданиями операционных систем часто встречается конструкция, инициирующая последовательное выполнение серии задач (программ), причем выходной поток каждой предшествующей задачи используется в качестве входного потока для следующей за ней. Эта конструкция позволяет, например, компактно и наглядно задать цепочку различных типовых преобразований данных, предназначенную для формирования печатного документа.

В монографии [77], написанной под несомненным влиянием концепций системы UNIX, предпринимается попытка применения цепочечного подхода к программированию задач вычислительной математики. Автор считает этот подход одним из средств достижения повторной используемости («переиспользуемости») программ. Все же основная масса крупных программных комплексов для решения вычислительных задач опирается на каркасный подход, который имеет не меньше оснований претендовать на роль инструмента для обеспечения повторной используемости. Относительно неширокая распространенность цепочечного подхода объясняется, по-видимому, тем, что накладываемые им суровые ограничения на управляющие связи модулей влекут за собой существенные трудности при программировании и определенные потери эффективности при выполнении программ.

В то же время для некоторых предметных областей, в частности, для электротехнических расчетов, применение

цепочечного подхода оказывается весьма плодотворным. Цепочечная схема удобна и при организации обработки результатов натурных или вычислительных экспериментов. Тут отдельные модули могут вычленять интересующие экспериментатора данные, различным образом преобразовывать их и выводить на различные устройства визуализации. Наконец, цепочечный подход полезен в задачах искусственного интеллекта, когда непосредственное программирование решаемой задачи по каким-либо причинам затруднено или невозможно и в функции пакета входит построение алгоритма решения на базе имеющихся в пакете модулей и их спецификаций.

Цепочечный подход обеспечивает безболезненность развития программного фонда. Если вновь написанный модуль соответствует требованиям аппарата формирования цепочек, то он легко воляется в коллектив написанных ранее модулей, поскольку появление его не повлечет за собой какой-либо переделки их текстов.

Простейшая конструкция языка заданий пакета при цепочечном подходе – перечисление входящих в цепочку модулей. Если же формируемая цепочка достаточно длинна или же явное перечисление модулей по каким-то причинам неудобно, то используются более развитые конструкции языка заданий, речь о которых пойдет в следующем разделе.

Здесь остается только заметить, что многообразие схем конфигурирования модулей в пакетах не исчерпывается каркасным и цепочечным подходами. Возможны и другие схемы, кроме того, используются смешанные каркасно-цепочечные формы. Однако рассмотренные два подхода представляются наиболее интересными, и именно поэтому они были выбраны для подробного анализа.

1.6. Планирование вычислений

В этом разделе мы остановимся на различных конструкциях языков заданий и средствах их системной поддержки, применяемых при цепочечном подходе к конфигурированию модулей функционального наполнения.

Как уже упоминалось, в простейшем случае язык заданий позволяет явно перечислить в необходимой последовательности все составляющие расчетную цепочку модули. Если при реализации этой конструкции используется техника препроцессирования или макрогенерации [50,78], то появляются дополнительные возможности для приближения лекси-

ки и синтаксиса языка заданий к формам, характерным для обслуживаемой предметной области.

В некоторых пакетах семантика операторов языка заданий определяется с помощью частично упорядоченных множеств модулей. Так, например, в системах AED, САП-2 и АПРОКС [79-81] каждому оператору языка заданий соответствует дерево подчиненности, в вершинах которого расположены функциональные модули, реализующие в фиксированных деревьях сочетаниях выполнение различных конкретизаций этого оператора. При составлении расчетной цепочки анализируется не только текст задания, но и дерево подчиненности обрабатываемого оператора, и в качестве звена цепочки включается только тот фрагмент дерева, который относится к данной конкретизации оператора. Здесь мы имеем дело с сочетанием каркасного и цепочечного подходов.

Однако, когда речь идет о цепочечном подходе, основной интерес представляют не упомянутые выше языковые конструкции, а возможности применения в основывающихся на этом подходе пакетах автоматического планирования вычислений. При автоматическом планировании на языке заданий записывается не цепочка модулей, а только непроцедурная постановка интересующей пользователя задачи. Системное наполнение пакета на основе такой непроцедурной постановки автоматически строит и выполняет необходимую для решения задачи цепочку модулей.

Напомним преимущества непроцедурной формы задания. Прежде всего, она позволяет формулировать задание более наглядно и компактно. Существенно расширяется круг потенциальных пользователей пакета, поскольку теперь от них не требуется знания каких-либо программистских подробностей решения задач. В то же время нередко пользователь способен самостоятельно спроектировать требуемую цепочку, но использует автоматическое планирование либо из соображений экономии усилий, либо из-за боязни допустить механическую ошибку, вероятность которой при применении непроцедурных форм существенно уменьшается.

Автоматическое планирование вычислений требует определенного усложнения функционального наполнения пакета. Теперь в нем размещаются не только программные тела модулей, но и знания об их семантике, а также знания об общих закономерностях предметной области. Эти знания могут быть представлены как в декларативной форме, т.е.

в виде системы утверждений и правил вывода, так и в процедурной форме, т.е. в виде набора программ.

Экспертное планирование [44,82-85]. Планировщик реализуется в виде иерархической подсистемы, состоящей из программы-администратора планирования и расширяемого набора специальных программ, называемых *экспертами*. Каждый эксперт ориентирован на составление плана вычислений применительно к конкретному типу контекстов задания или, другими словами, связан с вполне определенными задачами (подзадачами) из предметной области пакета. Работа планировщика подобна синтаксически управляемой трансляции: разбор текста задания осуществляется администратором, а семантическая переработка выделенных синтаксических конструкций, приводящая к построению плана вычислений, — экспертами. При этом основной цикл процесса формирования расчетной цепочки протекает следующим образом.

Администратор планирования, анализируя текст задания, выделяет в нем очередной планируемый расчетный контекст, определяет, какой эксперт должен обрабатывать этот контекст, и инициирует работу эксперта. Последний, используя сведения, находящиеся в информационной базе пакета, и указанные в задании параметры задачи, либо строит фрагмент расчетной цепочки, соответствующий рассматриваемому контексту задания, либо может, как это делается в пакете MEIWER [85], осуществить содержательное преобразование контекста, представив сформулированную в нем задачу в виде суперпозиции подзадач, подлежащих, в свою очередь, подобной обработке. После выполнения своих функций эксперт возвращает управление администратору.

В информационной базе пакета при экспертном планировании хранятся сведения о возможных алгоритмах решения той или иной задачи и допустимых значениях ее параметров, критерии корректности (применимости) алгоритма по исходным данным, вычислительные схемы алгоритмов и др. В качестве параметров задачи могут, например, указываться матрица и правая часть системы линейных алгебраических уравнений, точность задания их элементов, требуемая точность искомого решения, некоторые топологические свойства матрицы и т.д.

Следует отметить, что для организации экспертного планирования требуется сформулировать правила построения и выбора схем алгоритмов решения задач и на этой основе подготовить экспертов, которые по постановкам задач могут

сразу до конца составлять соответствующие расчетные цепочки.

Планирование на вычислительных моделях [6,86,87]. В теории и практике пакетной проблематики большое внимание привлекают методы планирования вычислений, основанные на использовании вычислительных моделей. Широкому признанию этих методов в значительной мере способствовали пионерские работы таллинской школы программирования, выполненные под руководством Б.Г.Тамма и Э.Х.Тыгу.

Вычислительная модель определяется как объект, состоящий из множества переменных и множества отношений, которые связывают эти переменные. Каждая переменная вычислительной модели соответствует некоторому содержательному понятию предметной области пакета и обладает именем и видом. Отношение выражает связь между значениями переменных и может использоваться для вычисления значений некоторых из них по известным значениям других переменных. Семантика каждого отношения задается модулем функционального наполнения пакета. Вычислительная модель может быть представлена либо в виде графа, определяющего информационные и логические связи между функциональными модулями, либо в виде совокупности утверждений о применимости и сочетаемости модулей.

В зависимости от сложности описания предметной области при разработке пакетов, реализующих рассматриваемую технику планирования, в информационную базу пакета заносят либо вычислительную модель предметной области, либо набор вычислительных моделей задач. Первый подход применяют в случае, когда предметная область (объект исследования) имеет относительно устойчивую структуру и может быть описана единой вычислительной моделью, на которой решаются все задачи пакета с различными вариантами исходных данных. Этот подход получил свое воплощение в пакетах, ориентированных на вычисления при проектировании крупных энергоустановок, технологических процессов [58].

Второй подход используется тогда, когда класс решаемых задач настолько разнообразен, что они не могут быть отражены в рамках одной вычислительной модели, или же реализация этих задач на такой единой модели сопряжена с неприемлемыми временными издержками. Построение своей вычислительной модели для каждой задачи дает возможность уменьшить объем данных, анализируемых в процессе планирования, что существенно повышает быст-

родействие планировщика. Кроме того, такая более гибкая по сравнению с единой моделью система представления знаний предоставляет возможность синтеза новых моделей (более сложных задач) на основе алгебры моделей [46,47,61].

Формирование расчетной цепочки на основе вычислительной модели для задачи вида «вычислить V_1, V_2, \dots, V_k по заданным величинам W_1, W_2, \dots, W_r », где все величины принадлежат предметной области, может выполняться различными способами.

Первый из них, называемый *методом прямой волны*, состоит в том, что первоначально в план включается один из модулей, входные величины которого принадлежат множеству заданных величин. При включении модуля в расчетную цепочку вычисляемые им величины добавляются к множеству заданных и исключаются из множества искомых (если они ему принадлежат). Далее процесс продолжается до тех пор, пока не будет исчерпано множество искомых величин или не станет ясно, что ничего нового вычислить уже нельзя [6,88].

Другой метод, называемый *методом обратной волны*, обеспечивает формирование плана вычислений в направлении от искомых величин к заданным, т.е. формирование расчетной цепочки идет справа налево. Сначала на модели выбираются и заносятся в цепочку модули, входные величины которых принадлежат множеству искомых. Далее в число искомых включают величины, необходимые для применения модулей, занесенных в план, и не принадлежащие множеству заданных. Величины, вычисляемые этими модулями, исключаются из числа искомых. Процесс продолжается до тех пор, пока множество искомых величин не будет исчерпано [58,89].

Как показывает практика работы с моделями, наилучшие результаты дает сочетание методов прямой и обратной волны [46].

Планирование на семантической сети [48,70,71,90]. Этот подход предполагает использование для организации автоматического планирования вычислений методов искусственного интеллекта. В первую очередь здесь имеются в виду системы представления знаний и методы поиска решений в пространстве состояний.

Семантическая сеть, используемая планировщиком, охватывает три раздела знаний: о предметной области; о

методах вычислений; о применимости методов вычислений и реализации этих методов.

Использование дедуктивных информационных систем [90-92], обладающих способностью осуществлять логический анализ большой базы данных и позволяющих при работе с ними применять приемы процедурного программирования, дает возможность накапливать указанные выше знания в трех видах: в виде фактов, правил вывода и в виде процедур.

Например, знания о предметной области могут быть представлены в информационной базе пакета в виде набора утверждений об объектах и правил вывода, в частности, правил, определяющих взаимосвязь между объектами предметной области. С другой стороны, знания о методах вычислений, применяемых в предметной области, могут быть заданы в виде процедур, т.е. модулей функционального наполнения. И, наконец, знания о применимости методов вычислений и реализации этих методов представляются в виде системы фактов и правил вывода, описывающих проблемную и системную семантику модулей.

Многоаспектность информации, содержащейся в семантической сети, позволяет качественно усовершенствовать сам процесс планирования. Наиболее существенными представляются следующие преимущества планирования на семантической сети.

Проблемная семантика модуля содержит сведения о том, как этот модуль может быть использован при расчетах, производимых в рамках рассматриваемой области. Важно, что в описании проблемной семантики внешняя идентификация параметров модуля осуществляется не через имена, а посредством фактов, описывающих свойства объектов предметной области, которые моделируются этими параметрами. Наличие таким образом представленной проблемной семантики модулей позволяет обеспечить большую гибкость планирования по сравнению с обсуждавшимися выше подходами, где сцепление модулей, образующих цепочку, опиралось на наименования их входов и выходов.

Системная семантика модулей содержит сведения о модуле как о процедуре, т.е. сведения о различных программных параметрах и эксплуатационных характеристиках. Доступность планировщику системной семантики модулей позволяет производить не просто планирование вычислений, а *оптимальное планирование*, причем с учетом различных критериев оптимизации. В частности, используя описание

системной семантики модулей, планировщик может составлять расчетные цепочки, которые дают минимальное суммарное время их выполнения или занимают минимальный по размерам участок памяти.

Используя семантическую сеть, можно формулировать и решать задачу построения оптимальной расчетной цепочки как задачу поиска минимального пути в пространстве состояний [93].

Под состоянием S будем понимать множество известных (т.е. заданных либо вычисленных) параметров модулей функционального наполнения. Каждый модуль m характеризуется в семантической сети именем, множеством входных параметров $in(m)$, множеством выходных параметров $out(m)$ и ценой $price(m)$. Модуль применим к состоянию S , если $in(m) \subseteq S$. Применение модуля m к состоянию S порождает новое состояние S' , причем $S' = S \cup out(m)$. Путем из состояния S_0 в состояние S_k называется цепочка модулей функционального наполнения (m_1, m_2, \dots, m_k) , последовательное применение которых к состоянию S_0 порождает последовательность состояний (S_1, S_2, \dots, S_k) , причем каждое из звеньев цепочки модулей удовлетворяет условию применимости $in(m_i) \subseteq S_{i-1}$. Сумма цен модулей, входящих в цепочку, называется ценой пути из S_0 в S_k .

Задачей алгоритма планирования является построение по заданному начальному состоянию S_{beg} и целевому состоянию S_{end} минимального по цене пути в некоторое состояние S' , такое, что $S' \supseteq S_{end}$. Данная задача может решаться методами перебора в пространстве состояний или модификациями этих методов [93, 94].

Используя в качестве цены модуля тот или иной его атрибут (время выполнения, требуемая оперативная память и т.п.), можно оптимизировать составляемую планировщиком программу по тому или иному параметру.

В зависимости от того, в какой период времени по отношению к вычислительному процессу, реализующему задание пользователя, работает планировщик, различают *статическое* и *динамическое* планирование. Статическое планирование осуществляется до выполнения вычислительного процесса и в большинстве пакетов реализуется как один из этапов обработки входной информации и генерации рабочей программы. Однако только статического планирования в ряде случаев оказывается недостаточно для эффективного исполнения вычислительного процесса. Тогда статическое планирование дополняется или подменяется динамическим

планированием, которое реализуется в ходе вычислительного процесса. Как правило, динамическое планирование основывается на технике интерпретации [46,58].

Проведенное рассмотрение методов организации планирования вычислений, разумеется, не отражает полностью всего того многообразия решений, которые встречаются в планировщиках развитых пакетов. Весьма перспективными представляются, например, схемы, сочетающие в себе преимущества различных методов планирования (в частности, планирование на семантической сети с использованием экспертов [95]), и особенно диалоговые формы построения расчетной цепочки (когда начальный план вычислений, заданный пользователем, в ходе диалога детализируется и уточняется планировщиком на основе знаний, заложенных в информационную базу пакета [44,96]).

1.7. Операционные возможности

Операционными возможностями пакета будем называть совокупность организационных, управляющих и сервисных функций, предоставляемых его системным наполнением. Эти функции призваны, с одной стороны, использовать в интересах пакета весь арсенал средств штатного программного обеспечения, а с другой — «спрятать» от пользователя рутинные механизмы этого обеспечения, или, по крайней мере, значительно упростить и унифицировать доступ к ним. В результате формируется специализированная операционная среда, обеспечивающая благоприятные условия для проведения вычислений в данном приложении.

Некоторые из задач, возникающих при реализации операционных возможностей пакета, являются традиционными в том смысле, что они встречаются при разработке всякой программной системы, имеющей собственный входной язык и надстраиваемой над штатным обеспечением; другие характерны именно для пакетов программ. Охарактеризуем кратко наиболее существенные из этих задач и рассмотрим применяемые способы их решения.

Межмодульный интерфейс. Межмодульным интерфейсом называется организация информационных (интерфейс по данным) и управляющих (интерфейс по управлению) связей между модулями. Интерфейс определяется принятым регламентом модуляризации функционального наполнения и системными средствами поддержки сопряжения модулей по данным и управлению.

Анализ практики разработки модульных проблемно-ориентированных систем и пакетов программ [34,58,65,74] позволяет выделить три формы межмодульного интерфейса по данным.

Наиболее простым с точки зрения системной реализации является жесткий интерфейс. Он предполагает, что модули программируются только в соответствии с регламентом, строго фиксирующим для каждого модуля структуру, местоположение и другие характеристики обрабатываемых данных. Этим обеспечивается полная информационная и программная совместимость тех модулей, которые могут сочетаться в различных расчетных конфигурациях. Системная поддержка жесткого интерфейса осуществляется штатными средствами редактирования межмодульных связей.

В случае полужесткого интерфейса разработка программных тел модулей также ведется в соответствии с некоторым регламентом, устанавливающим определенную совокупность допустимых структур и способов хранения данных. Вместе с тем каждому модулю предоставляется известная свобода выбора в рамках этой совокупности. Системная поддержка полужесткого интерфейса обеспечивается либо штатной системой управления базами данных, либо специализированной системой хранения данных, входящей в состав системного наполнения пакета. Такие системы предоставляют языковые и программные средства, с помощью которых задаются (в программных телах модулей) и реализуются процедуры интерфейса.

Отметим, что в случае как жесткого, так и полужесткого интерфейса системная адаптация программного материала осуществляется путем внутренней модуляризации, т.е. все положения регламента модуляризации должны быть соблюдены на этапе разработки программных тел модулей.

Гибкий интерфейс позволяет осуществить сопряжение по данным независимо запрограммированных модулей. Системная поддержка гибкого интерфейса, так же как и полужесткого, осуществляется штатной или специализированной системой хранения и преобразования данных. Однако принципиальное отличие системы с гибким интерфейсом состоит в том, что она предоставляет возможность внешней модуляризации, т.е. системной адаптации программного материала не при составлении программ, а на этапе включения их в функциональное наполнение пакета. Это обстоятельство позволяет быстро развивать функциональное наполнение пакета, вклю-

чая в него программы, характеризующиеся значительной информационной рассогласованностью.

При организации межмодульного интерфейса по управлению применяются схемы прямой и косвенной коммутации модулей, а также сочетание этих схем.

В случае прямой коммутации передача управления вызываемому модулю осуществляется непосредственно из тела вызывающего. Такого рода схема передачи управления обычно обеспечивается штатными средствами редактирования связей и широко применяется в тех случаях, когда структуры управления в программах, генерируемых пакетом, достаточно устойчивы и необходимо исключить временные издержки при передаче управления от модуля к модулю. Прямая коммутация применяется обычно при каркасном подходе к конфигурированию модулей.

Схема косвенной коммутации характерна для цепочечного подхода. Она предполагает, что управляющие связи между модулями устанавливаются при участии некоторого программного посредника. В качестве такого посредника может выступать штатный или специальный динамический загрузчик или монитор (управляющая программа) пакета. Косвенная коммутация применяется в тех случаях, когда управляющие связи между модулями заранее фиксировать нельзя, поскольку они возникают динамически в ходе вычислительного процесса.

В ряде пакетов применяется подход, сочетающий преимущества обеих схем коммутации [50,74]. При этом управляющая связь первоначально устанавливается по схеме косвенной коммутации, а затем поддерживается (пока это позволяют доступные ресурсы памяти) по схеме прямой коммутации. Оригинальная реализация этого подхода осуществлена в системе ИС-2 [51].

Исполнение расчетной цепочки. Для организации исполнения составленной планировщиком расчетной цепочки модулей применяется техника компиляции или интерпретации [4,18,58].

В первом случае процессы генерации программы и проведения расчета строго разделены по времени выполнения: сначала по заданной расчетной цепочке формируется задание транслятору с базового языка пакета и полностью генерируется рабочая программа, а затем выполняется один или несколько расчетов по этой программе. Использование режима компиляции имеет два неоспоримых достоинства. Во-первых, как известно, техника компиляции дает возмож-

ность получать эффективные по времени исполнения программы, и, во-вторых, эти программы могут отчуждаться от пакета и использоваться автономно. В связи с этим техника компиляции, как правило, применяется в пакетах, ориентированных на обеспечение массовых и длительных расчетов.

При использовании модулей в режиме интерпретации процессы генерации рабочей программы и расчета выполняются квазипараллельно под общим управлением монитора вычислительного процесса. Реализовать в пакете технику интерпретации, вообще говоря, существенно проще, чем компиляцию. Интерпретация часто позволяет сократить объем оперативной памяти, используемой пакетом при выполнении вычислений. Наконец, режим интерпретации расчетной цепочки весьма удобен, если решение задачи ведется в форме диалога.

В некоторых случаях целесообразно комбинировать технику компиляции и интерпретации при исполнении расчетной цепочки. При этом наиболее критичные с точки зрения времени выполнения участки расчетной цепочки компилируются, и интерпретируется уже не цепочка, а частично сгенерированная программа. Такой подход обеспечивает возможность достижения разумного компромисса между временем исполнения задания и требуемыми ресурсами оперативной памяти.

Хранение информации. Информация, которую приходится хранить и обрабатывать с помощью пакетов прикладных программ, весьма разнородна: программный материал, исходные данные расчетов, промежуточные данные расчетов (записи контрольных точек), результаты расчетов, документация. Среди средств системной поддержки хранения этой информации традиционно выделяются два крупных раздела: хранение программного материала и хранение расчетных данных [18, 34, 46, 58].

Хранение программного материала может быть организовано на базе штатных средств, используемых общецелевыми редакторами, трансляторами и загрузчиками. Однако в случаях, когда при создании рабочей программы используются развитые алгоритмы генерации программного текста, возникает необходимость в реализации специализированных средств хранения [40, 97].

Для организации хранения расчетных данных обычно используются возможности штатных файловых систем. Эти возможности могут быть дополнены, например, специализи-

рованными средствами буферизации данных в оперативной памяти, которые учитывают специфику вычислительных процессов приложения и позволяют тем самым существенно снизить временные издержки [18,98].

Вместе с тем противопоставление данных и программ в ряде приложений представляется весьма искусственным и, более того, препятствует эффективной организации работ.

Нередко при проведении серии расчетов требуется в каждом очередном варианте расчета изменить не только часть исходных данных, но и некоторый фрагмент алгоритма расчета. В этом случае при традиционном подходе для формирования требуемого варианта приходится обращаться к двум разнородным системам хранения, что сопряжено с известными неудобствами.

Сходные сложности возникают и при попытке связать некоторый вариант программы, хранящийся как программный материал, с результатами расчета по нему, хранящимися как данные. Тем самым, в частности, появляется препятствие на пути применения таких современных методов, как инкрементальный подход к поддержке разработки и отладки программ [99].

Наконец, множество неудобств доставляет автономное хранение программной документации. Если преодолеть эту автономию, можно было бы широко использовать ссылки из программ и данных на сопроводительные документы и наоборот.

Приведенные аргументы говорят в пользу единой системы для организации хранения всей информации. Помимо разрешения трудностей, подобных перечисленным выше, ее использование позволяет унифицировать средства доступа к разнородной информации и за счет концентрации усилий на одной системе хранения снабдить ее богатым набором возможностей. Следует отметить, что разработка подобных систем в силу специфики пакетов прикладных программ нередко выходит за рамки традиционной проблематики систем управления базами данных [46,47,100].

Общение пользователя с пакетом. Штатное программное обеспечение предоставляет обычно достаточно широкий (от операторов языка управления заданиями и директив операционной системы до диалоговых систем) набор средств для организации общения пользователя с пакетом [101]. Это позволяет при необходимости относительно легко обеспечить не одну, а несколько форм общения, отражающих содержание различных этапов работы пакета и, главное,

различную степень активности пользователя на каждом из этапов.

В пакетах программ реализуются следующие формы общения с пользователем или, другими словами, режимы выполнения заданий.

Наименее активное взаимодействие пользователя с пакетом происходит при закрытой обработке заданий. (Во избежание терминологической путаницы мы используем слово «закрытая» вместо слова «пакетная».) При таком режиме пользователь и на этапе подготовки задания, и на этапе его исполнения не имеет прямой связи с пакетом и все взаимодействие выражается в тех сообщениях и листингах, которые пользователь получает от пакета после успешного или неуспешного прохождения задания через вычислительную систему. Некоторые пакеты предоставляют определенную диалоговую среду, используемую на этапе подготовки задания, но сформированные с ее помощью задания выполняются все же в закрытом режиме. Закрытая обработка находит применение в пакетах программ, генерирующих рабочие программы с большим временем прогона (так называемые *фоновые программы*) [18,65,74].

В пакетах, предназначенных для обслуживания ограниченного набора сравнительно простых задач, часто используется режим «запрос-ответ». При таком режиме пользователь, находящийся за терминалом, оперативно получает результаты вычислений по формируемым им заданиям (запросам). В некоторых пакетах в режиме «запрос-ответ» можно выполнять несложные арифметические вычисления, знакомиться с возможностями пакета или с его текущим состоянием и т.д. [102,103].

Применяются и более совершенные формы диалога, позволяющие пользователю и пакету регулярно обмениваться информацией, уточняющей и направляющей работу пакета на всех этапах его функционирования. В частности, можно предоставить пользователю средства оперативного управления ходом выполнения расчетной программы, позволив ему в определенных узловых пунктах решения выбирать дальнейшее направление вычислительного процесса [17,44,104–107].

Сервис. Сервисные компоненты системного наполнения пакета, как и сервисные средства всякой развитой программной системы, обеспечивают пользователя инструментарием для оперативного решения вопросов, возникающих при эксплуатации, развитии и сопровождении пакета.

Одной из наиболее распространенных сервисных возможностей является редактирование исходных текстов модулей функционального наполнения пакета. Эта услуга особенно необходима при проведении вычислений в режиме активной дисциплины, когда пользователь, выполняя с помощью пакета сборку новой версии программного комплекса, должен предварительно модифицировать ряд модулей, входящих в этот комплекс. Для выполнения редактирования либо используются штатные универсальные редакторы, либо создаются специализированные средства, ориентированные на базовый язык (языки) пакета и позволяющие редактировать модули, не выходя за пределы среды задания для пакета [40,66,102].

Другой важный раздел сервисных возможностей охватывает функции отладки и наблюдения за процессом исполнения задания пользователя. Сюда входят, в частности, средства диагностики ошибок и выдачи сообщений, локализующих ошибки, средства прослеживания хода выполнения рабочей программы и избирательной распечатки ее управляющих параметров [50,63,64].

При работе с пакетами программ, функциональное и системное наполнение которых интенсивно развивается, возникает задача постоянного отслеживания в пользовательской документации всех вносимых в программы изменений. Для развивающихся пакетов на смену выпуску быстро устаревающих печатных руководств приходят сервисные средства оперативного документирования. Эти средства, во-первых, позволяют в любой момент получать печатные документы, отражающие текущее состояние пакета, и, во-вторых, предоставляют возможность простого и легкого внесения изменений в тексты существующих документов. Кроме того, они облегчают процесс подготовки текущих документов к публикации [102,108].

1.8. Информационное обеспечение

Трудно достигнуть высокого уровня автоматизации прикладной деятельности, не уделив должного внимания средствам информационного обслуживания. Хотя это обстоятельство и признается абсолютным большинством разработчиков пакетов, вопросы информационного обслуживания в пакетной проблематике проработаны недостаточно. По-видимому, это объясняется тем, что над разработчиками довлеют сложившиеся формы использования универсального программного обеспечения.

Однако пакеты как специализированные программные системы и по своей внутренней организации, и по назначению существенно отличаются от других общецелевых компонентов программного обеспечения и приносят разнообразные стили и формы ведения работ, весьма отличающиеся от «штатных». Что же касается документации, то хотя в этом аспекте к пакетам предъявляются повышенные требования [109–112], но обилие документации еще не гарантирует простоты и удобства использования пакета, и, кроме того, как уже отмечалось, в развивающихся пакетах публикуемая документация практически никогда не соответствует текущему состоянию пакета.

Информационное обеспечение является той составляющей системного наполнения, с помощью которой достигается «взаимопонимание» между пользователем и пакетом в целом, а также между отдельными компонентами пакета. Поэтому, когда мы говорим об информационном обеспечении пакета, то имеем в виду специализированное обслуживание двух видов: внешнее и внутреннее.

Целью внешнего информационного обслуживания является удовлетворение информационных запросов, поступающих от пользователей, т.е. проведение диалога, в течение которого пользователь получает или сообщает пакету сведения, необходимые для выполнения задания или поддержания работоспособности пакета. Другими словами, внешнее информационное обеспечение пакета мыслится как средство решения конечного числа так называемых информационных задач. В качестве примеров таких задач можно привести информационную задачу изучения программно-эксплуатационных характеристик модулей функционального наполнения пакета или задачу конструирования программного комплекса, реализующего конкретную расчетную постановку.

При реализации информационных задач широко применяется техника *меню* [46,96,113], когда содержание диалога регламентируется и направляется ориентированным на решение конкретной информационной задачи справочником, имеющим древовидную структуру. В данном случае активной стороной в диалоге является пакет, который, получив от пользователя ответы на сравнительно небольшое количество вопросов (перечень допустимых ответов на задаваемые вопросы представляется на экране в форме меню), выдает ему нужную справку или указания о том, как действовать в данный момент сеанса работы с пакетом.

При использовании техники *гипертекста* информация организуется в виде сети. В любой момент диалога на экране присутствует некоторая страница документации, в тексте которой выделены (например, подсветкой) узловые понятия. С помощью функциональных клавиш пользователь может произвольным образом перемещаться по сети: либо перейти к чтению следующей/предыдущей страницы, либо, указав интересующее его узловое понятие, перейти к просмотру относящихся к нему документов, либо вернуть на экран предыдущий документ и т.д. Для наглядного отображения документов широко применяются машинная графика и другие современные средства представления информации.

Меню и гипертекст используются обычно для решения информационной задачи ознакомления с возможностями и/или с текущим состоянием пакета. Для другой, не менее важной задачи – построения требуемой версии расчетной программы – требуются несколько иные диалоговые средства. Если при ознакомлении с возможностями пакета запоминается только путь, пройденный пользователем по сети документации, то для построения расчетной программы нужно выяснить и запомнить информацию о стоящей перед пользователем проблеме. Здесь в качестве активной стороны диалога приходится выступать как пользователю, так и пакету. Желательно, чтобы информационная служба пакета обладала способностью воспринимать в ходе подобного диалога умалчиваемый пользователем контекст точно так же, как это сделал бы квалифицированный специалист.

Информационное обеспечение пакета может поддерживать двусторонне активный диалог, руководствуясь сценарием решения конкретной информационной задачи. Такой сценарий можно задавать, например, в терминах продукций и вводить заранее в информационную базу пакета. Продукция представляет собой записанное на некоем языке правило, состоящее из двух частей. В первой из них содержится список фактов, характеризующих ту или иную ситуацию, возникающую в ходе диалога, а во второй – список действий, которые необходимо выполнить в этой ситуации. По мере усложнения информационной задачи такой сценарий, или система продукций, может наращиваться путем добавления отдельных продукций, причем это следует выполнять с учетом связи новых продукций со старыми [70,106,114].

Более универсальный подход к организации обоюдодиактивного диалога, допускающий, вообще говоря, обработку любого прагматически осмысленного сообщения или запроса

пользователя, предполагает совместное использование развитой информационно-логической системы и лингвистического процессора, приближающего язык общения к естественному для пользователя языку [90,91]. Отметим, что хотя язык общения с информационно-логической системой обладает довольно ограниченной лексикой и весьма жестким синтаксисом, все же он обычно достаточно легко воспринимается пользователем пакета. В силу этого информационно-логическая система может применяться не только как база для разработки развитого информационного обеспечения, но и как первичная форма его реализации, которая может использоваться самостоятельно [103].

С точки зрения программной организации средства обеспечения внешнего информационного обслуживания могут быть встроены в системное наполнение пакета или же разрабатываться относительно автономно. В последнем случае облегчается оформление информационных средств в виде надстройки над мощной общецелевой информационно-логической системой типа ДИЛОС [90], ВОПРОС-ОТВЕТ [91].

Внутреннее информационное обслуживание имеет своей целью удовлетворение запросов, поступающих от различных информационных компонентов пакета при его функционировании. Такие запросы возникают, например, при составлении плана вычислений, генерации программного комплекса, упорядочивании архива программного материала и банка расчетных данных и т.д. Для того, чтобы обеспечить столь разнородные системные процессы, внутренняя информационная служба пакета должна располагать знаниями о предметной области, семантике модулей функционального наполнения, вычислительной среде, в которой осуществляется эксплуатация пакета.

В заключение, отметим, что при разработке информационных средств пакета, какими бы они ни были, необходимо прежде всего сформировать терминологический базис, опираясь на который можно описать требуемые для организации информационного обслуживания знания. Такой базис, т.е. совокупность терминов, обозначающих объекты, их свойства и отношения между объектами, может быть сформирован в результате понятийного и лексического анализа автоматизируемой прикладной деятельности и основных структурных составляющих пакета в аспектах тех информационных услуг, которые должен обеспечивать пакет.

1.9. О последующих главах

Итак, рассмотрены основные архитектурные компоненты системного обеспечения пакетов прикладных программ. Практически все эти компоненты нашли свое воплощение в производственных пакетах, описываемых в последующих главах, которые и составляют основное содержание книги.

Дело в том, что главное в искусстве разработки пакетов программ – умение выявить и надлежащим образом удовлетворить средствами системного наполнения пакета специфические потребности автоматизируемой прикладной деятельности. Конечно, проведенные в настоящей главе рассмотрения могут подсказать разработчику решения ряда характерных проблем, однако не меньшую пользу можно извлечь и посредством изучения опыта построения конкретных пакетов, к изложению которого мы и переходим.

Г л а в а 2

ПРОГРАММИРОВАНИЕ ЗАДАЧ ВЫЧИСЛИТЕЛЬНОГО ЭКСПЕРИМЕНТА

2.1. Вычислительный эксперимент

Одним из важнейших классов задач, решаемых в настоящее время с помощью вычислительных машин, являются задачи вычислительного эксперимента. Вычислительным экспериментом называется метод изучения устройств или физических процессов посредством построения их математической модели и последующего численного исследования этой модели, позволяющего «проиграть» их поведение в различных условиях [1,3,14]. Программу, реализующую математическую модель, можно рассматривать как аналог опытной установки, используемой при натурном эксперименте. Численное исследование модели позволяет не только определять различные характеристики процессов, оптимизировать конструкции или режимы функционирования проектируемых устройств, но и обнаруживать новые процессы и свойства, о которых перед проведением вычислительного эксперимента исследователю ничего не было известно.

Вычислительный эксперимент занимает промежуточное положение между аналитическим исследованием и натурным экспериментом. При аналитическом исследовании также строится математическая модель, но модель настолько простая, что для ее изучения оказывается достаточно аналитических выкладок и не требуется привлечения численных методов, опирающихся на использование вычислительной техники. Аналитическое исследование применяется, если необходимо быстро получить грубые характеристики исследуемого процесса. При таком изучении реальных процессов приходится идти на серьезные упрощения, пренебрегая некоторыми закономерностями. И хотя для заданной модели аналитический подход нередко дает точное решение, тем не менее эта упрощенная модель обычно не позволяет получить требуемую в практических расчетах точность

оценок. Численные методы, применяемые в вычислительном эксперименте, дают возможность изучать существенно более сложные модели, достаточно полно отражающие исследуемые процессы.

Сравнивая вычислительный эксперимент с натурным (физическим), следует отметить два преимущества вычислительного эксперимента. Во-первых, его проведение, как правило, экономически дешевле проведения физического. Во-вторых, в ряде случаев натурное исследование сопряжено с измерением чрезвычайно большого числа параметров, созданием и поддержанием критических режимов и т.п., и поэтому вычислительный эксперимент нередко оказывается практически единственно возможным способом исследования.

Отмеченные достоинства вычислительного эксперимента вывели его в настоящее время в число основных методов исследования таких крупных физических и инженерно-технических проблем, как задачи ядерной энергетики, освоения космического пространства и др.

2.2. Цикл вычислительного эксперимента.

Обработка результатов

Можно выделить следующие этапы вычислительного эксперимента для решения физических задач [3] (рис.2.1).

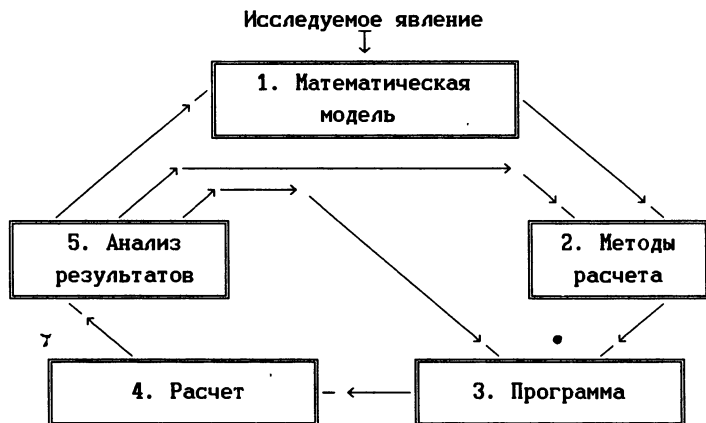


Рис 2.1 Этапы вычислительного эксперимента

1. Построение математической модели (составление уравнений, описывающих исследуемое явление).

2. Выбор численных методов расчета (построение дискретной модели, аппроксимирующей исходную математическую задачу, построение разностной схемы, разработка вычислительного алгоритма и т.д.).

3. Создание программы, реализующей вычислительный алгоритм.

4. Проведение расчетов и обработка полученной информации.

5. Анализ результатов расчетов, сравнение с натурным экспериментом.

Обычно на последнем (пятом) этапе выясняется, что необходимо внести определенные изменения в решения, принятые на этапах 1, 2 или 3. Так, может выясниться, что построенная модель недостаточно хорошо отражает особенности исследуемого явления. В этом случае модель корректируется, вносятся соответствующие поправки в численные методы и реализующие их программы и выполняется новый расчет. Таким образом, цикл вычислительного эксперимента воспроизводится в полном объеме. При анализе результатов могут быть выявлены какие-либо недостатки используемых численных методов, связанные, в частности, с соображениями точности или эффективности. Изменение методов влечет за собой изменение соответствующих программ и т.д. Иначе говоря, цикл повторяется в несколько сокращенном виде (этапы 2-5). Наконец, может оказаться неудачным некоторое программное решение, например, выбранный способ работы с внешней памятью. Пересмотр таких решений приводит к повторению этапов 3-5.

В настоящей книге основное внимание уделено этапу 3 - подготовке расчетных программ, и именно для автоматизации этого этапа предназначаются некоторые описываемые в последующих главах пакеты. Прежде чем перейти к развернутой характеристике задач этапа 3, остановимся кратко на особенностях обработки и анализа результатов расчетов на этапах 4 и 5.

Непосредственными результатами расчетов, выполняемых программами вычислительного эксперимента, являются обычно большие массивы данных. Если при анализе этих данных исследователь имеет дело только с напечатанными на бумаге бесконечными столбцами чисел, то труд его оказывается крайне неэффективным. На помощь тут могут прийти диалоговые формы ознакомления с результатами эксперимента, включающие, в частности, широкое использование возможностей машинной графики.

Приведем пример характерной задачи анализа результатов. Пусть в результате расчетов получена некоторая многомерная матрица большого объема, представляющая собой вычисленные значения некоторой функции нескольких переменных на прямоугольной сетке. Требуется, зафиксировав значения некоторых переменных, получить двумерную вырезку из этой матрицы и отобразить поведение определяемой ею функции двух переменных на графическом устройстве.

Эта конкретная задача представляется довольно простой. Неплохое ее решение в такой упрощенной постановке можно найти, например, в работе [77]. Однако попытка построения универсального инструментария для обработки реальных результатов вычислительного эксперимента наталкивается на труднопреодолимую преграду, связанную с многообразием форм представления результатов расчетов.

Здесь можно упомянуть, в частности, неравномерности сетки, на которой рассчитывается результирующая функция, специальные формы представления сильно разреженных матриц, различные способы задания особенностей функции и т.д. Использование специальных форм представления не позволяет применить штатные средства машинной графики непосредственно к результатам расчетов, так как эти средства ориентированы обычно на вполне определенный регулярный способ представления данных.

Поэтому построение инструментария для обработки результатов расчетов является нелегкой задачей. Необходимо выявить и систематизировать все используемые в конкретной автоматизируемой прикладной деятельности формы представления данных и для каждой из таких форм написать программу-преобразователь, приводящую эту форму к виду, воспринимаемому программами машинной графики. После этого уже не составляет особого труда создание языка, позволяющего пользователю по мере необходимости выделять и выводить на алфавитно-цифровые и графические устройства интересующие его подмножества результирующих данных.

2.3. Подготовка расчетных программ

Подготовка расчетных программ – один из наиболее трудоемких этапов вычислительного эксперимента. На каждом цикле вычислительного эксперимента выполняется построение новой версии программы (см. рис. 2.1), и системное наполнение любого пакета программ, ориентированного на автоматизацию такого рода деятельности, должно включать

в себя средства поддержки этой постоянно повторяемой процедуры.

С точки зрения программной реализации наиболее существенны такие особенности метода вычислительного эксперимента, как многомодельность и многовариантность. *Многомодельность* означает, что в ходе вычислительного эксперимента математическая модель многократно изменяется и уточняется, а *многовариантность* – что выполняемые уточнения модели могут затем комбинироваться в самых разнообразных сочетаниях.

Эти особенности проявляются в том, что, во-первых, основная работа по проведению вычислительного эксперимента приходится не на первое выполнение, а на многократное повторение описанного выше цикла. Данное соотношение в полной мере распространяется и на программу: подавляющая часть усилий разработчиков приходится не на создание первой версии расчетной программы, а на различные ее модификации, отражающие вновь принимаемые решения о модели, методах расчета и организации программы.

Во-вторых, решения (а вместе с ними и соответствующие им программы), пересматриваемые на очередном цикле вычислительного эксперимента, как правило, не отбрасываются и могут использоваться впоследствии для других расчетов. Например, метод, оказавшийся непригодным для одной математической модели, вполне может подойти для расчета следующей модели и т.д.

Таким образом, главное направление деятельности по программированию задач вычислительного эксперимента – не создание новых, а развитие существующих программ. Но это развитие осуществляется, как правило, не за счет замены существующих модулей их более совершенными версиями, а за счет включения в программный фонд все новых и новых модулей, отражающих различные решения, принимаемые в ходе эксперимента.

Такое накопление модулей, а также упоминавшаяся выше сложность реализуемых математических моделей приводят к тому, что размеры программного фонда становятся весьма внушительными. Например, для решаемых в ИПМ имени М.В.Келдыша АН СССР задач одномерной газовой динамики программный фонд содержит свыше 1000 написанных на Фортране модулей с суммарной длиной более $7 \cdot 10^4$ предложений, для задач радиационной защиты – около $2.3 \cdot 10^5$ предложений Фортрана. Фонд постоянно пополняется новыми модулями, которые в самых разнообразных комбинациях используются в расчетах.

Ориентироваться в столь обширном и сложном программном хозяйстве очень нелегко. Основным содержанием работ здесь становится даже не написание новых программных модулей, хотя и это постоянно имеет место, а конструирование из модулей расчетных программ («опытных установок») и решение целого ряда чисто технических проблем, связанных с развитием и сопровождением программного фонда. Выполнение таких работ на базе средств штатного программного обеспечения требует весьма значительных трудозатрат.

Можно указать следующие основные причины неудобства использования штатного программного обеспечения для программирования задач вычислительного эксперимента:

- прикладной программист в своей повседневной работе вынужден иметь дело со значительным числом относительно разнородных системных средств: библиотеками текстов и загрузочных модулей, редакторами, загрузчиками и т.д.;
- некоторые компоненты штатного программного обеспечения ориентированы в первую очередь на программы малых и средних размеров; непосредственное применение этих компонентов при программировании задач вычислительного эксперимента приводит к росту непроизводительных затрат;
- штатное программное обеспечение ориентировано на «обобщенного пользователя», т.е. стремится охватить возможно более широкий круг задач, и, следовательно, не должно, вообще говоря, включать в себя специальные средства, учитывающие особенности вычислительного эксперимента, связанные, в частности, с уже упоминавшимися многомодельностью и многовариантностью.

Отмеченные сложности организации программирования задач вычислительного эксперимента на базе средств штатного программного обеспечения закономерно подводят нас к мысли о создании специальных системных средств. Такие средства должны обеспечивать хранение, пополнение и модификацию программного фонда, а также простой, гибкий и быстрый механизм сборки из отдельных модулей фонда различных вариантов программ для конкретных расчетов.

Среди других задач, которые могли бы быть решены с помощью специальных системных средств, можно указать поддержку использования нестандартизированного программного фонда и информационное обслуживание.

Проведение вычислительного эксперимента нередко начинается не «с нуля», а в условиях, когда определенная часть программного фонда уже существует. Эту часть

составляют программы, созданные в ходе предшествовавших расчетных работ. Среди них могут встретиться нестандартизированные программы, т.е. программы, для которых не планировалось совместное использование в рамках решения одной сложной задачи. Системные средства могли бы облегчить использование в расчетах подобных программ.

Поскольку объем и сложность программного фонда весьма велики, проводящему эксперимент прикладному программисту приходится иметь дело с большим количеством различных характеризующих этот фонд данных. В частности, ему требуются сведения о составе и особенностях версий расчетных программ, использовавшихся на различных «витках» цикла вычислительного эксперимента, функционально-эксплуатационные характеристики отдельных входящих в фонд модулей, значения параметров, задававшиеся при проведении того или иного расчета, и т.д. Специальные системные средства могли бы, с одной стороны, обеспечивать пользователя информацией о составе и характеристиках программного фонда, а с другой стороны, – поддерживать целостность пары «программы – документация», стимулируя и облегчая коррекцию документации по мере внесения изменений в программы. Трудно переоценить важность такого рода услуг в среде вычислительного эксперимента, где непрерывно идет пополнение и изменение программного фонда.

И все же главная проблема программирования задач вычислительного эксперимента – организация конструирования расчетных версий программ. Одно из наиболее убедительных решений этой проблемы предлагает проект OLYMPUS, которому посвящен следующий раздел.

2.4. Проект OLYMPUS

История решения с помощью ЭВМ задач вычислительного эксперимента насчитывает уже свыше трех десятилетий. За это время накоплен весьма значительный опыт организации их программирования, позволяющий говорить о существовании технологии программирования задач вычислительного эксперимента. Многие особенности этой технологии были обобщены и развиты в проекте OLYMPUS [63,64].

Проект OLYMPUS представляет собой набор соглашений, регламентирующих оформление, документирование и организацию взаимодействия и использования программ. Можно выделить следующие основные цели, которые преследуют эти соглашения: повышение наглядности программ; упрощение

эксплуатации сложного программного фонда; расширение возможностей обмена программным материалом. Рассмотрим, какие средства предлагает проект OLYMPUS для достижения этих целей.

Повышению наглядности служат прежде всего соглашения, связанные с оформлением и документированием программ. В программах, разрабатываемых в рамках проекта OLYMPUS, строго фиксированы форма и место комментариев, определяющих назначение программы, используемые в ней алгоритмы и т.д. Предусмотрены средства наглядного представления внутренней рубрики программы, четко разграничивающие относительно самостоятельные ее фрагменты. Разработаны методы классификации всех применяемых в эксперименте программ. Существуют соглашения о форме используемых в программах идентификаторов: по первой букве идентификатора можно определить не только тип, но и некоторые функциональные характеристики соответствующей переменной.

Эти, а также некоторые другие соглашения по оформлению и документированию дают возможность пользователю, знакомому с проектом, легко ориентироваться в своих и чужих программах, без труда определять назначение любой программы, быстро отыскивать и разбирать необходимые ему фрагменты алгоритмов и т.д. Кроме того, разработаны средства системной поддержки документирования, позволяющие собрать в одном формируемом документе все комментарии определенного вида, «рассыпанные» по многочисленным программам. Тем самым тексты программ превращаются в основу документации, благодаря чему во многих случаях удастся избежать часто встречающейся рассогласованности между программами и сопровождающими их документами.

Вторая цель проекта OLYMPUS, наиболее существенная с точки зрения системного программного обеспечения вычислительного эксперимента, — упрощение эксплуатации сложного программного фонда. Она достигается с помощью соглашений об организации взаимодействия и использования хранящихся в фонде программ.

Эти соглашения могут служить убедительной иллюстрацией применения описанного в предыдущей главе каркасного подхода к конструированию программ из модулей. Программа, выполняющая конкретный расчет, моделирующий исследуемое физическое явление, представляется в виде фиксированного набора составных функциональных частей, образующих каркас программы, называемый здесь *схемой счета*. За каждой из таких частей закрепляется строго определенная

часть работы по расчету и характеризующее ее имя, называемое *функциональным именем*. Например, ENERGY – часть, обеспечивающая расчет энергии, STEPON – основной шаг вычислений и т.д. Кроме того, детально прорабатываются и фиксируются способы взаимодействия между выделенными таким образом частями. После этого для любой из функциональных частей может быть написано, вообще говоря, несколько различных реализаций, или модулей, из которых в любом конкретном расчете будет участвовать лишь один.

В результате переход к следующему витку цикла вычислительного эксперимента сводится, с точки зрения программной реализации, к подмене определенного модуля (модулей), использовавшегося на предыдущих витках. Если принятое на данном витке цикла решение реализуется впервые, соответствующий модуль должен быть написан заново; если же требуется применявшийся ранее метод, можно воспользоваться модулем, написанным для него в свое время и хранящемся в программном фонде.

Предлагаемая дисциплина работы требует, безусловно, определенных трудозатрат, связанных с тщательной разработкой каркаса программы на ранней стадии программирования вычислительного эксперимента. Но эти трудозатраты в полной мере окупаются простотой последующего расширения и использования программного фонда. Ведь развитие фонда идет здесь безболезненно, не затрагивая текстов написанных ранее программ!

Третья цель проекта OLYMPUS – расширение возможностей обмена программным материалом между различными лицами и организациями, занимающимися задачами вычислительного эксперимента в одной и той же или близких областях. Дело в том, что до появления этого проекта в качестве объектов обмена выступали, в основном, либо большие законченные программы, решающие конкретную задачу или класс задач, либо подпрограммы и составленные из них библиотеки, реализующие отдельные математические методы. Применение соглашений OLYMPUS дает возможность обмениваться практически произвольными фрагментами программ, конечно, при условии, что и производитель, и потребитель используют один и тот же каркас. Так, например, объектом обмена могут быть модули, обеспечивающие ввод и начальную обработку исходных данных, расчет уравнения энергии, и вообще реализации любых частей, предусмотренных каркасом. При этом изменения, требующиеся для адаптации полученных частей к уже существующим у потребителя

программам, минимальны и не затрагивают программ потребителя.

Простота адаптации достигается прежде всего за счет применения всеми участниками обменов программным материалом единого каркаса программы – схемы счета. Кроме того, все работающие в системе соглашений OLYMPUS программы пишутся на диалекте Фортрана, воспринимаемом большинством существующих трансляторов. Наконец, части программ, связанные с взаимодействием с конкретной операционной системой, фиксированным образом выделяются и документируются.

Проект OLYMPUS получил широкое международное признание. Журнал *Computer physics communications* регулярно публикует алгоритмы, разработанные в рамках этого проекта, которые используются многими организациями в различных странах, в том числе в СССР.

Соглашения OLYMPUS существенно облегчают проведение вычислительного эксперимента и при отсутствии средств системной поддержки. Но с появлением таких средств отдача этих соглашений многократно возрастает. Это утверждение в полной мере можно отнести как к повышению наглядности программ, так и к облегчению конструирования расчетных вариантов.

Мы уже упоминали ряд системных разработок, выполненных авторами проекта OLYMPUS и ориентированных на то, чтобы сделать программы более наглядными, облегчить процессы ознакомления с программным фондом и составления документации. В то же время расчетные варианты программ конструировались авторами проекта на базе штатных системных средств. Поэтому для формирования вариантов им приходилось использовать универсальный текстовый процессор, составляя для каждого расчета задания размером до нескольких сотен строк.

Соглашения OLYMPUS поддерживает и описываемый в следующей главе пакет прикладных программ Сафра. Однако в этом пакете, в отличие от системных разработок авторов OLYMPUS, на первый план выдвигается задача автоматизации конструирования программ вычислительного эксперимента.

Г л а в а 3

СИСТЕМА САФРА: КОНСТРУИРОВАНИЕ ПРОГРАММ ВЫЧИСЛИТЕЛЬНОГО ЭКСПЕРИМЕНТА

3.1. Технология конструирования

Сафра [115] – Система Автоматизации Физических РАсчетов – предназначена для поддержки конструирования программ вычислительного эксперимента. Ключевой проблемой вычислительного эксперимента с точки зрения программирования является организация многочисленных изменений программ, отражающих различные решения, принимаемые в ходе эксперимента. Такие решения могут быть связаны и с модификацией исследуемой математической модели, и с выбором того или иного численного метода, и с пересмотром некоторых элементов алгоритма. Но в любом из этих случаев изменение решений неизбежно влечет за собой изменение программы.

Принимаемые в ходе эксперимента решения обычно не являются окончательными, нередко приходится возвращаться к отвергнутым на ранних стадиях эксперимента моделям, методам или алгоритмам. Поэтому изменения программ связаны в основном не с заменой того или иного модуля его улучшенной версией, а с пополнением программного фонда новыми модулями, различные сочетания которых образуют расчетные программы. Поскольку в процессе эксперимента часто требуется повторение выполнявшихся ранее расчетов, желательно организовать работу таким образом, чтобы постоянные изменения программного фонда, связанные с появлением новых модулей, проходили безболезненно, т.е. не затрагивали текстов написанных ранее программ.

Поясним проблему безболезненного внесения изменений на примере. Пусть с помощью вычислительного эксперимента исследуется поведение некоторого устройства в зависимости от материала, из которого оно изготовлено. Пусть решено на начальной стадии эксперимента в качестве материала использовать сталь или медь, а затем исследо-

вать какие-либо другие материалы. Будем считать, что появление того или иного материала приводит к определенным изменениям в конструкции устройства и поэтому для исследования каждого нового материала требуется переписать некоторый фрагмент моделирующего алгоритма.

Как следует организовать программу моделирования, если мы стремимся минимизировать изменения существующих частей программы при появлении нового материала и соответственно нового фрагмента алгоритма?

На первый взгляд, неплохо выглядит следующее решение. В том месте программы, где необходимо выполнять различные действия в зависимости от использования того или иного материала, записывается (к примеру, на Паскале) оператор выбора следующего вида:

```
...
case материал of
    сталь: < моделирование стали > ;
    медь:  < моделирование меди  >
end { case }
...
```

Тогда, присваивая с помощью исходных данных расчета переменной «материал» значения «сталь» или «медь», можно переключать алгоритм моделирования с одного материала на другой. С точки зрения организации последующих изменений основной результат такого построения заключается в том, что в программе неявно, но достаточно четко определено место для размещения новых фрагментов алгоритма, моделирующих новые материалы: такие фрагменты записываются в виде новых ветвей оператора выбора.

Рассмотренное решение можно упрекнуть за то, что в нем непроизводительно расходуется оперативная память, поскольку в программу включаются все имеющиеся ветви выбора, а фактически в любом расчете используется лишь одна из них. Однако такой упрек можно довольно легко парировать – достаточно превратить оператор выбора в конструкцию периода компиляции.

Главный же технологический изъян применения в подобной ситуации конструкции выбора несколько менее очевиден. Он заключается в том, что появление новой ветви выбора неизбежно влечет за собой редактирование текстов существующих программ, что, как известно, нередко приводит к потере их работоспособности.

Совершенно иное решение этой задачи, основанное на каркасном подходе и описанных в предыдущей главе согла-

шениях проекта OLYMPUS, предлагается в системе Сафра. На месте оператора выбора в программе записывается специальная конструкция INSERT (вставить) периода компиляции вида

. . .
INSERT материал
. . .

Теперь фрагменты алгоритма, соответствующие различным материалам, оформляются в виде самостоятельных текстовых объектов и помещаются в специализированный архив под соответствующими именами: «сталь», «медь» и т.д. Для формирования расчетной программы, моделирующей поведение устройства, изготовленного из определенного материала, следует задать соответствие между именем, записанным в операторе INSERT («материал»), и именем, под которым хранится соответствующий фрагмент алгоритма (например, «сталь»):

материал = сталь

Сафра построит программу, заменив оператор INSERT указанным фрагментом.

С точки зрения технологичности внесения последующих изменений приведенное решение принципиально отличается от рассматривавшегося выше использования оператора выбора. Здесь появление фрагмента алгоритма, соответствующего новому материалу, не вызывает редактирования уже имеющихся текстов программ. Новый фрагмент просто помещается в архив под уникальным именем, что никак не может повлиять на работоспособность других хранящихся там программ.

Оператор INSERT – не единственная конструкция Сафры, поддерживающая «безболезненные» изменения программ. Если позволяют соображения наглядности и эффективности, то все элементы набора фрагментов алгоритма, соответствующие альтернативным решениям, могут быть оформлены в виде подпрограмм. Тогда вместо специального оператора INSERT периода компиляции в программе записывается обычный оператор вызова подпрограммы периода выполнения. Например, если программа пишется на Фортране, она будет выглядеть так:

CALL МАТЕРИАЛ

Все альтернативные фрагменты алгоритма представляют собой подпрограммы с одним и тем же именем МАТЕРИАЛ, однако в специализированный архив они помещаются под различными именами: «сталь», «медь» и т.д. Далее с точки зрения пользователя Сафры формирование расчетных программ выполняется точно так же, как и для оператора INSERT, — указывается соответствие между именем подпрограммы и именем, под которым хранится в архиве ее реализация. Например:

материал = сталь

На большинстве вычислительных установок, в частности, на ЭВМ БЭСМ-6, для которой разрабатывалась Сафра, штатные системные средства не позволяют организовать эффективное использование описанной технологии. Дело в том, что точек ветвления, подобных рассмотренной в примере, в реальных программах насчитывается обычно несколько десятков. Для того, чтобы произвести все необходимые манипуляции с текстами программ и организовать их последующие трансляцию и исполнение, прикладной программист вынужден выполнять множество рутинных операций. Сафра избавляет его от этой скучной и утомительной деятельности, существенно повышая тем самым производительность труда при программировании задач вычислительного эксперимента.

Точнее говоря, описанная технология конструирования явилась отправной точкой для цикла работ, посвященного автоматизации конструирования программ вычислительного эксперимента и приведшего к созданию системы Сафра. В последующих разделах приводятся краткие сведения об организации этих работ и описываются основные функциональные возможности Сафры.

3.2. Организация и содержание работ по созданию Сафры

Сафра создавалась в рамках исследований по пакетам прикладных программ, проводимых в Институте прикладной математики имени М.В.Келдыша АН СССР под руководством академика А.А.Самарского. Первая производственная версия системы была реализована на ЭВМ БЭСМ-6 в среде операционной системы Диспак и мониторной системы Дубна в 1978 году.

В создании Сафры постоянно принимали участие две группы программистов: прикладные и системные. Прикладные программисты были заняты созданием функционального наполнения пакетов, для которых средства, предоставляемые Сафрой, служили в качестве языка заданий и системного наполнения. Системные программисты разрабатывали и совершенствовали средства, обеспечивающие хранение функционального наполнения (архив) и интерпретацию языка заданий, т.е. части, непосредственно не связанные с предметной областью.

Конечно, все наиболее важные решения, касающиеся, например, расширения возможностей языка заданий, принимались на основе совместных обсуждений двух групп разработчиков. Тем не менее каждая из групп обладала известной автономией, не вникая, в частности, в менее существенные детали деятельности другой группы.

Предоставляемые Сафрой средства ориентированы в первую очередь на разработчиков больших программ для задач вычислительного эксперимента. Такие программы, создаваемые с помощью Сафры, характеризуются следующими параметрами: общее число модулей программного фонда – около 1000, суммарная длина программ – около 50 тысяч предложений, число гнезд каркаса расчетных программ, используемых для формирования вариантов, – свыше 50.

До появления Сафры столь обширное программное хозяйство приходилось вести в основном на базе средств штатного программного обеспечения. Это приводило к тому, что основная доля усилий по проведению вычислительного эксперимента уходила на чисто техническую работу, связанную с организацией хранения и использования написанных ранее программ. Для проведения расчета прикладной программист должен был найти и подготовить тексты используемых модулей, оттранслировать их, сформировать необходимую для данного расчета библиотеку объектных модулей и т.д.

Определенную помощь в поиске и подготовке текстов программ оказывала система PATCHY [40], идеология которой близка к проекту OLYMPUS. Сафра заимствовала из PATCHY некоторые технические решения, но пошла существенно дальше, взяв на себя, в частности, не только подготовку текстов, но и проблемы автоматизации трансляции и сборки. Технические трудности, возникающие при организации трансляции и сборки, поясним на примерах из практики программирования на ЭВМ БЭСМ-6.

В одной библиотеке объектных модулей мониторинной системы Дубна нельзя хранить несколько модулей с одним и

тем же именем (имя служит в качестве ключа модуля в библиотеке). Но если при работе по технологии, описанной в предыдущем разделе, определено, что некоторый альтернативный фрагмент алгоритма оформляется в виде подпрограммы, то все различные реализации этого фрагмента будут иметь вид подпрограмм с одним и тем же именем. Поэтому прикладной программист вынужден для каждой такой реализации заводить отдельную библиотеку. Число таких библиотек катастрофически растет, что приводит к неэкономичному расходованию внешней памяти и усложнению процесса сборки расчетных вариантов.

Другой пример связан с экономией времени трансляции. Для того, чтобы основную массу необходимых для очередного расчета модулей не транслировать заново, а брать готовыми из библиотеки объектных модулей, необходимо постоянно поддерживать целостность пар «исходный текст модуля – его объектный код». Для этого при внесении изменений в текст модуля (или же в текст альтернативного фрагмента алгоритма, вставляемого в модуль посредством оператора INSERT) необходимо позаботиться об уничтожении старой версии объектного модуля в библиотеке. Если текст модуля включает в себя несколько независимых вставок (INSERT), то ручное выполнение такого отслеживания становится крайне затруднительным.

Сафра полностью избавляет своих пользователей от забот, связанных с трансляцией. Конструирование программ ведется исключительно на уровне исходных текстов, от пользователя скрыты получение, хранение и использование результатов трансляции. Трансляция и сборка напоминают о себе только в случае обнаружения ошибок. Обеспечивается компактное хранение объектных модулей, автоматическое поддержание соответствия между исходными текстами и объектными модулями, минимизация времени перетрансляции за счет автоматического выявления при сборке всех готовых объектных модулей.

Особенностью процесса конструирования программ является его тесная связь с многочисленными компонентами штатного системного программного обеспечения: редакторами, трансляторами, загрузчиками и т.д. Успех попытки автоматизации конструирования во многом определяется тем, насколько удачно разрабатываемые средства сочетаются с штатной системной средой. При создании Сафры пришлось потратить немало усилий для того, чтобы процесс конструирования не выглядел инородным телом в среде штатного обеспечения.

Кроме того, с точки зрения пользователя известные трудности вызвала попеременная работа с разнородными штатными средствами. По мере своего развития Сафра брала на себя решение все более широкого круга проблем, связанных с обработкой прикладных программ, сопроводительных описаний и других материалов функционального наполнения. В результате сформировался набор операций, в значительной мере экранирующий прикладного программиста от особенностей операционной системы, трансляторов, загрузчиков и т.д.

Решение перечисленных выше проблем позволило превратить Сафру в эффективный инструмент разработки программ вычислительного эксперимента. В ходе многолетней эксплуатации системы только в Институте прикладной математики были сформированы и выполнены десятки тысяч различных расчетных вариантов программ для задач математической физики. Достижение такой производительности труда прикладных программистов без помощи Сафры вряд ли было бы возможно.

С появлением промышленной версии Сафры контингент пользователей системы расширился, включив в себя целый ряд внешних организаций. В связи с этим разработчикам Сафры пришлось вплотную столкнуться со всем «многоцветьем» версий системного программного обеспечения ЭВМ БЭСМ-6. Потребовались специальные меры для того, чтобы Сафра «научилась» настраиваться на несколько независимо развиваемых ветвей операционной системы Диспак, мониторной системы Дубна, на диалоговые редакторы Димон и Краб.

Заслуживает упоминания и разработка специализированного архива Сафры, предназначенного для хранения исходных текстов программ, объектных модулей, сопроводительной документации и других материалов функционального наполнения. При реализации Сафры одним из наиболее узких мест оказалась эффективность организации хранения упомянутых материалов во внешней памяти. Поэтому многое зависело от решения вопроса о том, воспользоваться ли для организации хранения каким-либо существующим внешним по отношению к Сафре инструментом или же затратить усилия на создание собственного специализированного архива.

Какие соображения можно привести в пользу специализированного архива? В работе [100], например, убедительно показано, что традиционные СУБД ориентированы в первую очередь на организационно-хозяйственные применения и

мало пригодны в задачах обработки результатов физических экспериментов. Есть все основания для того, чтобы распространить данное заключение и на область хранения программных материалов. Напротив, специализированный архив способен удовлетворить специфические, зачастую достаточно жесткие требования, связанные, в частности, с эффективностью формирования расчетных задач.

Кроме того, Сафра постоянно развивается, выдвигая все новые запросы к средствам хранения. Поэтому даже если некоторая СУБД представлялась пригодной на ранней стадии реализации Сафры, всегда существовала опасность, что в дальнейшем появление новых запросов вынудит отказаться от нее.

И все же решение о реализации собственного специализированного архива пришло к разработчикам Сафры не сразу. Первая, макетная версия Сафры использовала для хранения данных средства системы УПД-6.

Но затем, когда выяснилось, что объем работ по реализации специализированного набора средств хранения составит не более 15–20% от объема работ по реализации Сафры в целом, было решено, что такую цену можно заплатить за уверенность в достижении любого требуемого уровня эффективности. Эта уверенность может появиться лишь в случае, когда сами разработчики имеют возможность модифицировать средства хранения по мере появления новых требований.

Разработанный архив позволил получить необходимые прикладным программистам показатели эффективности. Создание, уничтожение или модификация модуля программного архива требуют не более одной секунды процессорного времени БЭСМ-6. Сборка наиболее сложных вариантов программ (использующих до 100 модулей из архива размером порядка 4 Мбайт) требует 15–20 секунд. Такие накладные расходы представляются вполне допустимыми, поскольку, например, трансляция всех программ сложного варианта занимает 10–15 минут времени процессора.

3.3. Архитектура

По современным представлениям архитектура (т.е. представляющийся пользователю внешний вид) реализации Сафры на ЭВМ БЭСМ-6 выглядит несколько архаично. Дело в том, что ввиду скромных диалоговых возможностей вычислительной среды БЭСМ-6 Сафра была ориентирована на пакетный режим работы. Однако, как показал опыт реализации Сафры

на ЕС ЭВМ, основная часть операций пакетной Сафры очевидным образом отображается на диалоговый режим. Поэтому содержащееся в данном и последующих разделах описание возможностей пакетной Сафры позволяет получить определенное представление и о диалоговой версии Сафры. О специфике диалоговой Сафры будет идти речь лишь в случае, если диалоговый режим выполнения какой-либо операции добавляет новые интересные возможности.

Итак, все виды работ, связанных с подготовкой и запуском программ на очередном витке вычислительного эксперимента, при использовании системы Сафра оформляются с помощью языка заданий пакета. Каждое задание состоит из нескольких последовательно выполняемых пунктов, соответствующих отдельным операциям. Например, существуют операции, позволяющие:

- записать (STORE) в архив пакета новый модуль;
- исключить (DELETE) модуль, записанный ранее;
- отредактировать (EDIT) или
- распечатать (PRINT) тексты модулей, хранящихся в архиве;
- распечатать каталог (CATALOG) модулей архива;
- переписать архивный модуль в файл редактора Димон (DIMON) или Краб (CRAB);
- собрать и выполнить (EXECUTE) конкретный расчетный вариант;
- распечатать (BOOK) в двухстраничном формате с редактированием тексты сопроводительных документов и т.д.

Первая группа операций (STORE, DELETE, EDIT, PRINT, CATALOG) предназначена для модификации или распечатки содержимого архива пакета. С помощью операции STORE в архив может быть записан под указанным именем произвольный текст. Любой текстовый объект, записанный в архив посредством STORE, называется *модулем*, а имя, под которым он хранится, — *архивным именем* модуля. Помимо архивного имени каждому модулю приписывается еще и *тип*, определяющий его назначение (например, PRUNIT — программная единица, допускающая автономную трансляцию, VARIANT — вариант расчета и т.д.). Операции DELETE, EDIT, PRINT, CATALOG будут описаны в последующих разделах.

Определенной слабостью пакетной версии Сафры является отсутствие собственного диалогового редактора для хранящихся в архиве текстов. Если требуется отредактировать текст архивного модуля, можно либо воспользоваться

встроенной в Сафру операцией пакетного редактирования (EDIT), либо связаться с одним из двух наиболее распространенных на БЭСМ-6 диалоговых редакторов – с Димоном (DIMON) или с Крабом (CRAB).

Заминку перед началом сеанса диалогового редактирования, связанную с необходимостью переписи обрабатываемого текста из архива в файл диалогового редактора, можно было бы устранить, если бы архив Сафры был построен на базе средств файловых систем этих редакторов. Однако такое решение, как отмечалось в п.3.2, повлекло бы за собой слишком большие потери эффективности при выполнении других операций Сафры. Собственный диалоговый редактор, который мог бы полностью решить данную проблему, не был включен в Сафру-БЭСМ-6 из-за большой трудоемкости его реализации. Отметим, что в диалоговой Сафре-ЕС удалось подключиться к штатному редактору СВМ практически без каких-либо потерь эффективности.

Если следовать технологии конструирования программ вычислительного эксперимента, описанной в п.3.1, то для задания конкретной расчетной программы каждому из вариантов гнезд ее каркаса следует соотнести некоторый реализующий его модуль архива (вставку или подпрограмму). Так как гнезда каркаса программы идентифицируются функциональными именами (в соответствии с соглашениями проекта OLYMPUS), а модули – архивными именами, для задания конкретной программы достаточно записать соответствующее число пар <функциональное имя> = <архивное имя>. В тексте задания такие пары записываются в пункте EXECUTE, обеспечивающим сборку и выполнение расчетного варианта, в виде MATCH-предложений

MATCH <функциональное имя> = <архивное имя>

Часто повторяющиеся группы MATCH-предложений могут быть оформлены и записаны в архив в виде специальных модулей типа вариант (VARIANT). В этом случае, для того чтобы включить в пункт EXECUTE такую группу, достаточно указать только имя модуля-варианта.

Наряду с модулями, непосредственно используемыми при формировании расчетных программ, в архиве пакета хранятся также тексты сопроводительных документов. С помощью операции BOOK такие тексты могут быть распечатаны на АЦПУ с редактированием в удобном для последующего использования двухстраничном формате. Кроме того, правила оформления текстов согласованы с системой АСПИД, позво-

ляющей автоматизировать подготовку документов к публикации.

3.4. Архив

Как уже упоминалось, архив системы Сафры служит для хранения программного фонда и сопроводительных документов. Текстовые объекты, записанные в архив, называются *текстовыми модулями*. С каждым модулем связываются ключ, под которым он хранится в архиве, – архивное имя, а также тип, определяющий назначение модуля, даты его создания, последнего изменения, использования, шифр пользователя, создавшего модуль, число строк в тексте модуля, объем памяти, занимаемый текстом, и другая информация. Допускаются следующие типы модулей.

MACRO – текстовая вставка. В виде модулей типа MACRO оформляются фрагменты программ, заданий, тексты сопроводительных документов.

PRUNIT – программная единица. Текст программы на алгоритмическом языке, допускающий автономную трансляцию.

VARIANT – вариант расчета. Группа соответствий (MATCH-предложений) между функциональными именами гнезд каркаса программы и архивными именами модулей.

В последующие версии Сафры предполагается включить модуль типа GROUP – группа архивных модулей. Применяются два способа формирования группы: *перечислительный* и *ассоциативный*. При перечислительном формировании архивные имена входящих в группу модулей явно перечисляются в тексте соответствующего модуля типа GROUP. При ассоциативном формировании явное перечисление архивных имен отсутствует, а в группу включаются модули, в которых указан атрибут принадлежности к группе. Группы позволяют рассматривать архив «под разными углами зрения», высекая подмножества модулей различной тематической направленности. С их помощью существенно компактнее записываются групповые операции распечатки, уничтожения, переписи во внешний архив, в файлы диалогового редактора и т.д. Группы используются также при реализации защиты от несанкционированного доступа.

Помимо текстовых модулей в архиве хранятся также *объектные модули*. Получение и использование объектных модулей полностью автоматизировано и скрыто от пользователя Сафры, работающего только на уровне исходных текстов программ.

Реализуется данная возможность следующим образом. Формирование текста расчетной задачи, рассмотренное в предыдущем разделе, выполняется посредством подстановки на место гнезд каркаса (задаваемых операторами INSERT) текстов необходимых архивных модулей. Единственное отклонение от этого правила возникает, когда подставляемый модуль имеет тип PRUNIT, т.е. представляет собой программную единицу, допускающую автономную трансляцию. В этом случае в процессе сборки выясняется, не выполнялась ли когда-либо трансляция данной программной единицы (с учетом конкретного набора текстов, подставляемых на место вложенных операторов INSERT).

Если трансляция уже выполнялась, подстановка текста модуля типа PRUNIT не производится, а из архива Сафры извлекается и включается в сборку полученный в свое время (при трансляции) соответствующий объектный модуль. Если же программная единица ранее не транслировалась, то в формируемую задачу включается ее текст для трансляции, причем полученный в результате объектный модуль не только используется в данной сборке, но и запоминается в архиве для последующих применений.

3.5. Дисциплина работы

В п.3.1 была рассмотрена технология конструирования программ вычислительного эксперимента, поддерживаемая системой Сафра. Определим теперь дисциплину работы, позволяющую в полной мере использовать преимущества данной технологии.

Напомним прежде всего основные технологические моменты применяемого в Сафре каркасного подхода к конструированию программ. Непосредственному программированию в вычислительном эксперименте предшествует этап функциональной модуляризации, результатом которого является разбиение исследуемой проблемы на ряд функциональных частей. Такое разбиение определяет каркас программы, гнезда которого соответствуют выделенным функциональным частям. Каждому гнезду каркаса присваивается имя (функциональное имя), а также фиксируются способ его программного оформления (автономно транслируемая программная единица или фрагмент текста на алгоритмическом языке с заданным набором переменных и т.д.) и способ организации его взаимодействия с другими гнездами.

Затем переходят к непосредственному программированию, в ходе которого для каждого из гнезд может быть написа-

но, вообще говоря, несколько программных реализаций. При проведении конкретного расчета каждому гнезду сопоставляется какая-либо реализация соответствующей функциональной части, в результате чего формируется расчетный вариант программы.

При модуляризации помимо общеизвестных соображений (функциональная самостоятельность, минимизация связей и т.п.) в случае вычислительного эксперимента на первый план выдвигается выделение частей, подверженных изменениям под влиянием изменяющихся в ходе эксперимента факторов. На каждом витке вычислительного эксперимента пересматриваются некоторые решения: могут измениться фрагмент математической модели, используемый метод расчета, организация таблицы, требования к точности решения и т.д. Если при функциональной модуляризации все эти изменяющиеся факторы были учтены и им соответствуют отдельные гнезда каркаса, то перестройка программы требует минимальных усилий.

Наиболее предпочтительным является соотношение «один фактор — одно гнездо каркаса». К сожалению, иногда не удается добиться односвязности всех частей алгоритма, подверженных влиянию некоторого фактора. Например, точность выполняемых расчетов может зависеть от реализации нескольких далеко отстоящих друг от друга фрагментов текста программы, объединение которых в одном модуле резко усложнило бы межмодульные связи. Здесь возникает соотношение «один фактор — несколько гнезд каркаса», а каждому значению изменяющегося фактора будет соответствовать не один, а несколько реализующих модулей, заполняющих эти гнезда.

Сафра в известной мере поддерживает многосвязные реализации факторов, позволяя объединять все относящиеся к одному значению фактора соответствия вида «функциональное имя» = «архивное имя» в один модуль типа вариант. При формировании расчетных программ следует ссылаться на этот вариант, а не выписывать явно все соответствия. Такой подход поможет избежать ошибок, связанных с появлением в программе модулей, относящихся к различным значениям одного и того же изменяющегося фактора.

Недопустимым представляется соотношение «несколько факторов — одно гнездо каркаса». Тут кроется источник дублирования программных текстов и значительного усложнения процессов развития программного фонда и сборки расчетных вариантов программ. Такое построение каркаса

программы свидетельствует обычно о небрежности, проявленной на начальной стадии ее разработки.

Последнее соображение нередко пытаются опровергнуть, задавая вопросы, подобные следующему: «почему нельзя сначала построить вариант программы, выбирая из программного фонда наилучшие с точки зрения времени выполнения модули, а затем, опираясь на тот же каркас и тот же программный фонд, построить программу, отдавая предпочтение модулям, запрограммированным Ивановым?»

Вообще говоря, такое построение вполне возможно и при однофакторных гнездах каркаса. В программе могут быть выделены части, в наибольшей степени влияющие на скорость выполнения, которые программируются в двух вариантах: в быстром, записанном на ассемблере, и в более медленном, но мобильном, записанном на языке высокого уровня. Кроме того, могут существовать, скажем, две методики вычислений, одна из которых предложена и реализована Ивановым, а другая – Петровым, хотя для именования методик лучше использовать не фамилии разработчиков, а функциональные характеристики, по которым они отличаются.

Чтобы получить быстрый вариант программы, здесь достаточно сослаться на модуль-вариант с архивным именем **БЫСТРЫЙ**, в котором содержатся указания (**МАТЧ**-предложения) о включении в собираемую программу всех ассемблерных компонентов. При этом все остальные реализации вариантных гнезд, в том числе и методика вычислений, будут выбираться по умолчанию. Для расчетов по методике Иванова следует задать соответствие

МАТЧ МЕТОДИКА = ИВАНОВ

в результате чего в программу будет включен соответствующий модуль с архивным именем **ИВАНОВ**.

Однако в приведенном выше вопросе обычно подразумевается несколько иная конструкция программы, противоречащая однофакторному построению гнезд каркаса. Предполагается, что любой модуль программного фонда может иметь произвольный набор атрибутов (например, **БЫСТРЫЙ ИВАНОВ** и т.д.) и каждый из этих атрибутов может, вообще говоря, служить инструментом конфигурационного управления.

Например, для одного и того же гнезда каркаса Сидоров мог написать на ассемблере модуль с атрибутами (**БЫСТРЫЙ, СИДОРОВ**), а Иванов – на Фортране модуль с атрибутами (**МОБИЛЬНЫЙ, ИВАНОВ**). В первую требуемую конфигурацию программы будет включен первый из модулей из-за наличия

у него атрибута БЫСТРЫЙ, а во вторую – второй из-за атрибута ИВАНОВ.

Закономерно возникает вопрос: а каково же назначение заполняемого этими модулями гнезда каркаса? Неясно, служит ли оно для разрешения противоречия быстрый-мобильный или же для того, чтобы дать возможность проявиться творческой индивидуальности Иванова и Сидорова. Такое построение размывает функциональную структуру программы, чрезвычайно затрудняя ее изучение и последующее развитие.

Следует подчеркнуть, что возражения вызывает вовсе не наличие у одного модуля нескольких разнородных атрибутов. Атрибутов может быть сколь угодно много. В частности, один модуль может одновременно принадлежать нескольким различным группам (см. п. 3.4), и это оказывает существенную помощь в процессе анализа содержимого архива. Однако для управления конфигурацией при каркасном подходе должен использоваться только один изначально выделенный атрибут модуля. В Сафре таким единственным атрибутом является архивное имя модуля.

Отметим также, что в рассмотрении вопросов конфигурационного управления нередко участвуют понятия «поколение» и «поставка», связанные с жизненным циклом модулей и программы в целом. При проведении вычислительного эксперимента потребность в анализе жизненного цикла возникает относительно редко, и поэтому в Сафре отсутствуют какие-либо средства поддержки этих понятий.

При освоении системы Сафра новыми пользователями известные трудности вызывает то обстоятельство, что существующие средства штатного программного обеспечения, как правило, не поддерживают сосуществование нескольких различных вариантов программы. При работе с таким штатным обеспечением пользователь привыкает получать новые варианты посредством редактирования текстов программ. Редактирование выполняется даже в том случае, когда определенно известно, что впоследствии вновь потребуются исходный текст. Очевидно, что данная дисциплина получения вариантов излишне трудоемка и чревата внесением ошибок при редактировании.

Сафра поддерживает сосуществование нескольких вариантов программы. Все части программы, меняющиеся от варианта к варианту, выделяются и оформляются в виде самостоятельных модулей. Различные реализации выделенных частей, соответствующие разным вариантам, сосуществуют в

архиве Сафры, причем в любой расчетной программе используется только одна из таких реализаций.

Отметим, что аппарат вариантов применим и при отладке программ. Задавая соответствия между функциональными и архивными именами, можно легко сгенерировать программу, составленную из любой требующейся для отладки комбинации имитирующих («заглушек»), отлаживаемых и отлаженных модулей.

Развитие программного фонда вычислительного эксперимента идет безболезненно до тех пор, пока его модификации связаны с факторами, изменение которых было предугадано при исходной модуляризации. Если же требуется изменить программу в новом, неожиданном направлении, то такое изменение может повлечь за собой некоторую перестройку программы.

В определенных случаях можно было бы обеспечить безболезненность и такой перестройки. Пусть изменение некоторого фактора не было первоначально предусмотрено, но при анализе существующих программ выяснилось, что это изменение влечет за собой всего лишь замену одного ограниченного фрагмента исходного текста алгоритма. Тогда можно было бы применить операцию «вынесение во вставку», параметрами которой являются координаты выносимого фрагмента текста и имя создаваемой вставки. В результате выполнения такой операции в архиве появляется новый модуль с требуемым фрагментом текста алгоритма, а на его месте в исходном тексте размещается соответствующий оператор INSERT.

Выполнение операции «вынесение во вставку», с одной стороны, гарантированно не нарушает работоспособности всех предшествующих вариантов программы, а с другой стороны, дает возможность безболезненно включить в программный фонд иную реализацию вынесенного фрагмента. Полезно иметь и обратную операцию «вставить текст модуля», выполняющуюся в случае, когда потребность в вариантности некоторого фрагмента отпала или же при выполнении операции «вынесение во вставку» границы фрагмента были выбраны неудачно. К сожалению, обе эти операции в Сафре пока не реализованы.

Еще один существенный с точки зрения дисциплины работы с Сафрой момент заключается в том, что тексты сопроводительных описаний хранятся в архиве наряду с текстами собственно программ. Это позволяет практически одновременно вносить изменения и в программы, и в сопроводительную документацию, в результате чего докуме-

итация постоянно отражает текущее состояние программного фонда.

3.6. Язык заданий

Работа с системой Сафра ведется посредством выполнения заданий, написанных пользователем на языке заданий. Задание состоит из совокупности пунктов задания, выполняемых последовательно и определяющих такие действия, как запись и исключение модулей из архива, редактирование текстов хранящихся в архиве модулей, сборка и запуск на счет вариантов программы, некоторые вспомогательные операции (распечатка и перфорация текстов модулей, распечатка каталога архива и др.).

Каждый пункт задания состоит из заголовка пункта задания, либо из заголовка пункта и тела пункта задания. Концом тела пункта служит заголовок следующего пункта. Заголовок пункта представляет собой строку следующего вида:

<признак> <операция> <операнды>

Признак начинается с первой позиции строки. Он является указателем управляющего предложения Сафры (например, заголовка пункта). Признаком может служить любая последовательность длиной не более трех символов, составленная из следующих символов: «+», «-», «*», «/». Признак не должен содержать пробелов. В пределах одного задания используется обычно один и тот же признак управляющего предложения, хотя, вообще говоря, он может быть изменен для произвольной группы пунктов задания с помощью операции FLAG (см. п.3.9). В последующих примерах в качестве признака будет, как правило, применяться символ «+». Возможность использования различных признаков введена для того, чтобы можно было отличить предложение языка заданий от других предложений в рамках кодировки, принятой в той или иной вычислительной системе. Так, например, в мониторной системе Дубна символ «*» в первой позиции используется для обозначения управляющих карт системы.

Операцию может отделять от признака произвольное число пробелов (в частности, ни одного). Операция задает действие, выполняемое соответствующим пунктом.

Операнды отделяются от операции и друг от друга одним или несколькими пробелами или запятыми. Вид и назначение операндов зависят от операции. Например, операндом операции DELETE является имя модуля, который нужно исклю-

чить из архива, у операции E (конец задания) операнд не задается.

Архивное имя, являющееся операндом во многих операциях, представляет собой идентификатор длиной не более чем шесть символов.

Тело пункта – это последовательность предложений языка заданий либо другой текст, заключенный между заголовком данного пункта и заголовком следующего пункта. Так, например, тело пункта STORE содержит предложения текста модуля, записываемые в архив. В теле пункта PRINT перечисляются имена модулей, тексты которых нужно распечатать на АЦПУ.

Конец задания. Концом задания служит пункт задания, заголовок которого имеет вид

<признак> E

Тело пункта E пусто. В одном задании может содержаться не более одного пункта E. Если пункт E пропущен, задание выполнится корректно, но в конце листинга будет напечатана строка

***END FILE**

C-предложения, размещаемые в любом месте задания в качестве комментария, имеют вид

<признак> C <текст>

где <текст> отделяется от символа C одним пробелом или запятой и может содержать любые символы.

Комментарии служат для пояснения отдельных конструкций задания и не оказывают никакого влияния на его выполнение. Комментарии, встретившиеся в теле пункта STORE, предназначенного для записи в архив нового модуля, запоминаются и воспроизводятся при распечатке текста данного модуля, но исключаются из модуля при сборке конкретного варианта программы.

Протокол задания. Протоколом выполнения задания является его листинг, печатающийся на АЦПУ. Если в процессе анализа некоторого пункта выявляется ошибка, в листинге печатается соответствующее сообщение и в зависимости от серьезности ошибки выключается выполнение данного и части последующих пунктов. Для каждого из невыполненных пунктов печатается сообщение:

!!!! БЫЛИ ОШИБКИ - ПУНКТ НЕ ВЫПОЛНЕН

Если в результате сбоя аппаратуры был испорчен носитель, на котором расположен архив, то при обнаружении этого факта будет выдано соответствующее сообщение и выполнение всех последующих пунктов будет прекращено.

Если ошибок не обнаружено, то после каждого выполненного пункта печатается сообщение о результате его работы. Например,

+++++ НОВЫЙ МОДУЛЬ DUMMY СОЗДАН

или

+++++ СФОРМИРОВАНА ЗАДАЧА ШИФР 031102

3.7. Операции над модулями

Запись в архив нового модуля (STORE). Для записи в архив Сафры нового модуля служит операция (пункт задания), заголовок которой имеет вид

<признак> **STORE** [<тип>:]<архивное имя>

где <тип> – тип записываемого модуля (в этой нотации квадратные скобки означают необязательность наличия их содержимого). Допускаются следующие значения типа: **MACRO** – текстовая вставка, **PRUNIT** – программная единица, **VARIANT** – вариант расчета. Если <тип> опущен, создается модуль типа **MACRO**. <архивное имя> – архивное имя модуля, т.е. имя, под которым модуль записывается в архив.

Телом пункта **STORE** служит текст создаваемого модуля. Так, например, в результате выполнения пункта **STORE** вида

```
+STORE PRUNIT:DUMMY
      SUBROUTINE DUMMY
      RETURN
      END
```

в архиве появится модуль **DUMMY** типа **PRUNIT**, состоящий из трех предложений.

На листинге задания все предложения тела пункта **STORE** нумеруются, что позволяет использовать листинг при последующем пакетном редактировании текста модуля. Вслед за последним предложением тела на листинге печатается сообщение о результате выполнения пункта.

Если в момент выполнения пункта STORE в архиве уже содержится модуль с данным архивным именем, то записи нового модуля не произойдет, в листинге задания будет напечатано соответствующее диагностическое сообщение, например

!!!!!! В АРХИВЕ УЖЕ ЕСТЬ МОДУЛЬ DUMMY

и выполнение задания будет продолжено.

Если все же требуется записать в архив под данным именем новый текст модуля, старый модуль необходимо предварительно исключить посредством операции DELETE (см. ниже).

Модуль, записанный в архив с помощью пункта STORE, может быть использован затем в следующих за этим пунктом операциях того же задания или же в последующих заданиях. Например, текст записанного модуля может быть распечатан с помощью операции PRINT, отредактирован с помощью операции EDIT, использован при формировании программы для конкретного расчета в операции EXECUTE и т.д. Все эти операции ссылаются на модуль посредством его архивного имени.

При записи модуля в архив посредством пункта STORE иногда приходится сталкиваться с ситуацией, когда нежелательно или невозможно записать текст одного или нескольких предложений модуля в том виде, в котором они будут использованы при формировании заданий для расчетов. Так, в частности, поскольку запуск заданий для пакета Сафра осуществляется на базе средств мониторной системы Дубна, в тексте задания не могут встречаться служебные предложения Дубны *NAME или *END FILE, так как мониторная система воспримет их не как фрагмент задания, а некоторым специальным образом.

Однако подобные предложения часто необходимо использовать при формировании заданий для расчетов. Поскольку задания формируются исключительно из текстов хранящихся в архиве модулей, нужно иметь возможность записи в архив текста модуля, содержащего указанные предложения. Этой цели служат S-предложения вида

<признак> S <произвольный текст> (1)

где <произвольный текст>, содержащий «скрываемое» предложение, отделяется от символа S одним пробелом или запятой.

S-предложения хранятся в архиве в виде (1) и распечатываются в таком же виде при выполнении операции PRINT,

но при формировании расчетных вариантов на место S-предложения подставляется только записанный в нем <произвольный текст>.

Исключение (DELETE) модуля из архива системы Сафра осуществляется с помощью пункта DELETE, заголовок которого имеет вид

<признак> DELETE [<тип>:] <архивное имя>

где <тип> – тип исключаемого модуля (если <тип> опущен, модуль исключается независимо от его типа); <архивное имя> – архивное имя исключаемого модуля. Тело пункта DELETE пусто.

Результатом выполнения пункта DELETE является исключение из архива модуля с указанным архивным именем, о чем в листинге задания печатается соответствующее сообщение. Например,

**+DELETE DUMMY
+++++ МОДУЛЬ DUMMY УНИЧТОЖЕН**

Если в архиве отсутствует модуль с указанным архивным именем, то выполнение задания будет продолжено и сообщение будет, например, таким:

!!!!!! В АРХИВЕ ОТСУТСТВУЕТ МОДУЛЬ DUMMY

Печать текста модуля (PRINT). Тексты модулей, хранящихся в архиве системы, можно распечатать с помощью пункта PRINT, заголовок которого имеет вид

<признак> PRINT <список архивных имен>

где <список архивных имен> – архивные имена модулей, текст которых должен быть распечатан на АЦПУ.

Если список достаточно велик, не поместившиеся в заголовке имена перечисляются в предложениях MODULE, составляющих тело пункта PRINT. Предложение MODULE имеет вид

MODULE <список архивных имен>

Архивные имена, входящие в <список архивных имен>, отделяются друг от друга пробелами или запятыми.

Предложения MODULE используются в телах пунктов PRINT (распечатка), PUNCH (перфорация), DIMON и CRAB (перепись в Димон и в Краб), BOOK (документирование) и SELECT (перепись из другого архива). Число предложений MODULE в

теле одного пункта не ограничено, причем предполагается, что список имен продолжается при появлении нового предложения MODULE в теле данного пункта.

Если какие-либо модули из перечисленных в списке отсутствуют в архиве, то вслед за телом соответствующего пункта печатается диагностическое сообщение с перечислением всех таких модулей, а распечатка текстов остальных модулей производится обычным образом.

В результате выполнения пункта PRINT формируется (в смысле ОС Диспак) задача печати, о чем на листинге задания печатается сообщение. Например,

```
+PRINT
MODULE PERSO, COMBAS
MODULE РАСЧЕТ
+++++ СФОРМИРОВАНА ЗАДАЧА ШИФР 031101000002
```

Сформированная задача печатает тексты модулей в том порядке, в котором они перечислены в пункте PRINT. Перед каждым текстом печатается заголовок, где указывается архивное имя модуля, его тип, шифр записавшего его пользователя, дата последнего изменения модуля и использованный в нем признак управляющего предложения. Все предложения текста модуля для удобства последующего пакетного редактирования снабжаются номерами. Так, текст модуля DUMMY из приведенного выше примера был бы распечатан в следующем виде:

```
МОДУЛЬ DUMMY ТИП PU ШИФР 1 ИЗМЕН. 10.12.87 ПРИЗНАК +
1 SUBROUTINE DUMMY
2 RETURN
3 END
```

Перфорация модуля (PUNCH). Тексты модулей, хранящихся в архиве системы, можно выдать на перфокарты с помощью пункта PUNCH, заголовок которого имеет вид

<признак> PUNCH <список архивных имен>

где <список архивных имен> – архивные имена модулей, текст которых должен быть отперфорирован. Если список не помещается в одной строке, архивные имена перечисляются в предложениях MODULE, составляющих тело пункта PUNCH.

В результате выполнения пункта PUNCH формируется (в смысле ОС Диспак) задача перфорации, о чем на листинге печатается соответствующее сообщение.

Отперфорированный текст нередко требуется затем записать в тот или иной архив Сафры. Поэтому перфокарты,

соответствующие одному перфорируемому модулю, представляют собой готовый фрагмент задания для системы Сафра, обеспечивающий запись в архив текста модуля. Первый пункт фрагмента – DELETE – задает уничтожение хранящегося в архиве старого модуля. Затем с помощью пункта STORE в архив записывается текст модуля. Заключающий фрагмент пункт FLAG позволяет записывать в рамках одного задания несколько модулей с различными признаками.

Пакетное редактирование текстов модулей (EDIT). Операция EDIT предназначена для пакетного редактирования текстов модулей, хранящихся в архиве пакета Сафра. Операция предоставляет для редактирования относительно скромный набор средств: она позволяет добавлять в текст модуля новые строки, исключать строки и заменять строки исходного текста. Заголовок операции имеет вид

<признак> **EDIT** [<тип>:]<архивное имя>

где <архивное имя> – имя модуля, который будет редактироваться; <тип> – тип этого модуля.

Телом пункта является задание для редактирования текста модуля, имя которого задано в заголовке пункта. Задание для редактирования состоит из предложений ABS, ADD, EXCL, REPL, TAKE, а также вновь создаваемых строк текста.

В результате выполнения пункта EDIT в архив записывается отредактированный модуль под тем же именем, которое задано в заголовке пункта. В случае успешного завершения редакции будет напечатано соответствующее сообщение, например

+++++ МОДУЛЬ DUMMY ОТРЕДАКТИРОВАН

Кроме того, будет обновлена дата последнего изменения модуля, которую можно узнать, распечатав модуль (PRINT) или выдав каталог архива (CATALOG). Если в задании на редактирование обнаружены ошибки, то в листинге печатается соответствующее сообщение, анализ правильности задания для редактирования продолжится, однако само редактирование произведено не будет, т.е. модуль останется неизменным.

Номер строки. В предложениях редактирования используется понятие номера строки. Номер строки представляет собой либо десятичное целое число из диапазона [0:K], где K – число строк редактируемого модуля, либо символ E, который можно использовать вместо номера последнего

предложения модуля. Все предложения считаются перенумерованными, начиная с 1 с шагом 1. Номера строк текста печатаются на листинге при записи модуля в архив (STORE), при его распечатке (PRINT) или перфорации (PUNCH). Именно в терминах этих номеров производится редактирование текста. Номер строки 0 используется только для добавления новых строк в начало текста.

Диапазон строк имеет вид

<номер строки 1> [-<номер строки 2>]

Если <номер строки 2> опущен, то предполагается, что диапазон включает только одну строку <номер строки 1>. Номера строк разделяются знаком «минус». Не требуется, чтобы номера строк, задаваемых в редактирующих действиях, располагались в порядке возрастания или убывания, но запрещается использовать пересекающиеся диапазоны строк. При обнаружении пересечения будет напечатано сообщение, например

!!!! ДИАПАЗОНЫ 135-155 И 140-144 ПЕРЕСЕКАЮТСЯ

Следует иметь в виду, что редактирующие действия выполняются по отношению к номерам строк исходного текста. Изменение номеров строк в результате редактирования должно учитываться только при последующих применениях к данному модулю операции EDIT.

Добавляемые строки задаются в операциях ADD и REPL. Добавляемые к исходному тексту строки образуют группы. Концом группы служит либо очередное редактирующее предложение, либо начало следующего пункта задания.

Предложение TAKE. Группы добавляемых строк, помимо явно заданных строк текста, могут содержать предложения TAKE, предназначенные для добавления строк текста, расположенных в данном или в других модулях архива. Предложение TAKE имеет вид

<признак> **TAKE** [<тип>:]<архивное имя> [<диапазон>]

и означает, что на его место подставляются строки модуля с заданным архивным именем из указанного диапазона. Если диапазон не задан, то подставляется текст всего модуля. Подстановка не будет произведена, если не совпадают признаки модуля, который редактируется, и модуля, строки которого подставляются. В этом случае выдается сообщение

!!!!!! НЕСОВПАДЕНИЕ ПРИЗНАКОВ

Предложение ADD предназначено для добавления в текст модуля новых строк и имеет вид

<признак> **ADD** <номер строки>

Добавляемые строки текста, следующие непосредственно за предложением ADD, будут вставлены в текст редактируемого модуля вслед за строкой, задаваемой в заголовке предложения.

Приведем фрагмент тела пункта EDIT с предложениями ADD.

```
+EDIT MOD
+ADD 37
      PRINT 10
      10  FORMAT ('END OF SOLUTION')
+ADD 38
      CALL SAVE
+ADD E
C -----
+TAKE ЧАСТЬ2
C -----
+TAKE ЧАСТЬ3 5-E
      END
+ADD 0
*FORTRAN
```

Первое предложение ADD задает добавление двух строк в текст модуля MOD после строки 37. Второе – добавление одной строки после строки 38. По третьему предложению в конец редактируемого модуля добавляется группа строк, составленная из заданных явно текстов комментариев, полного текста модуля ЧАСТЬ2 и строк текста модуля ЧАСТЬ3, начиная с пятой. Четвертое предложение задает вставку перед текстом модуля одной строки.

Предложение EXCL служит для удаления строк из текста модуля и имеет вид

<признак> **EXCL** <диапазон>

Из текста редактируемого модуля удаляются строки из указанного диапазона.

Предложение REPL служит для замены (т.е. удаления группы строк и добавления на их место других строк) и имеет вид

<признак> REPL <диапазон>

Строки из указанного диапазона удаляются, и на их место подставляются добавляемые строки, следующие непосредственно за предложением REPL. Число удаляемых строк может не совпадать с числом добавляемых. Более того, добавляемые строки могут отсутствовать, и в этом случае предложение REPL задает просто удаление строк. Можно сказать, что предложение REPL совмещает действие двух предложений: ADD и EXCL.

Предложение ABS. Одна из наиболее неприятных ошибок, возникающих при пакетном редактировании, — повторное выполнение одних и тех же редактирующих действий над одним модулем. Она возникает, в частности, в случае повторного выполнения задания, к которому обычно прибегают операторы в случае сбоя машины. Для предупреждения ошибок такого рода в Сафре принято соглашение, согласно которому повторное применение к модулю пункта EDIT с тем же самым телом рассматривается, вообще говоря, как некорректное и не выполняется. При этом печатается сообщение, например

!!!! МОДУЛЬ MOD УЖЕ РЕДАКТИРОВАЛСЯ

Если все же повторное редактирование необходимо, надо записать в теле соответствующего пункта EDIT предложение ABS вида

<признак> ABS

Печать каталога архива (CATALOG). Каталог архива пакета Сафра может быть распечатан с помощью пункта CATALOG, заголовок которого имеет вид

<признак> CATALOG

Тело пункта CATALOG пусто.

В результате выполнения этого пункта в листинге задания вслед за заголовком пункта CATALOG в алфавитном порядке имен печатается список хранящихся в архиве модулей. Каждому модулю соответствует одна строка каталога. В ней, кроме архивного имени, печатаются следующие данные: тип модуля; шифр пользователя, создавшего модуль; признак, с которым создан модуль; число строк в модуле; число блоков памяти или количество слов памяти БЭСМ-6, занимаемых модулем; дата создания модуля; дата и

время последнего изменения (редактирования) модуля; дата последнего использования модуля; 40 первых символов первого предложения текста модуля, которые могут служить неплохим комментарием, уточняющим назначение модуля.

Кроме того, распечатываются дата печати каталога и координаты архива на внешнем носителе. В конце выдается сводная информация о состоянии архива: его размер, число свободных блоков и т.д.

Такая организация распечатки каталога удовлетворяла пользователей Сафры, пока архивы содержали порядка 200–300 модулей. Однако со временем число модулей в архиве росло, и средний архив Сафры стал содержать около 1000 модулей. Текст каталога такого архива занимает свыше двух метров бумаги АЦПУ и неудобен для многих видов повседневной работы. Требуется предоставить пользователю возможность распечатки различных подмножеств каталога, задаваемых условиями, накладываемыми на атрибуты модулей, а также возможность переупорядочивать каталог по любому из атрибутов. Включение этих возможностей в пакетную версию Сафры потребовало бы значительно усложнения языка заданий, и поэтому они были реализованы только в диалоговой Сафре.

Перепись текстов модулей в Димон (DIMON) и в Краб (CRAB). Как уже упоминалось, диалоговое редактирование текстов хранящихся в архиве модулей выполняется внешними по отношению к пакетной версии Сафры средствами – с помощью диалогового монитора Димон или системы Краб. Редактирование предполагает выполнение следующих шагов.

Шаг 1. Посредством описываемых ниже операций DIMON и CRAB тексты модулей переписываются в указанный пользователем файл Димона или на сбойные зоны Краба.

Шаг 2. Переписанные тексты редактируются средствами Димона или Краба.

Шаг 3. Отредактированные тексты переписываются в архив Сафры путем запуска из Димона или Краба задачи переписи.

С точки зрения пользователя Сафры наибольшие неудобства вызывает заминка с началом редактирования, вызванная необходимостью выполнения шага 1. Хотя шаг 1 выполняется и не всегда, поскольку тексты находящихся в работе модулей обычно присутствуют уже и в архиве, и в редакторе, тем не менее наличие встроенного диалогового редактора, позволяющего избавиться от выполнения этого шага, является одним из важнейших преимуществ диалоговой версии Сафры.

Выполнение шага 3 осуществляется сравнительно легко, поскольку тексты переписываемых модулей оформляются, как и в операции перфорации PUNCH, в виде готовых фрагментов задания для записи в архив Сафры. В случае переписи в файл Димона эти фрагменты дополнительно обрамляются управляющими предложениями Димона «**///*****», позволяющими легко выделять тексты отдельных модулей.

Для переписи в файл Димона служит пункт DIMON, заголовок которого имеет вид

<признак> DIMON <идпол>: <файл> <список архивных имен>

где **<идпол>** – идентификатор пользователя Димона; **<файл>** – имя файла Димона, в который переписываются модули Сафры; **<список архивных имен>** – имена переписываемых модулей. Список имен может быть продолжен с помощью предложений MODULE.

В результате выполнения пункта DIMON формируется задача, которая дописывает тексты модулей (дополняемые указанным выше образом) один за другим в «хвост» указанного файла Димона.

Аналогично, для переписи модулей на сбойные зоны Краба служит пункт CRAB, заголовок которого имеет вид

<признак> CRAB <список архивных имен>

где **<список архивных имен>** – имена переписываемых модулей. Список может быть продолжен с помощью предложений MODULE.

Формируется задача, которая переписывает на сбойные зоны Краба тексты модулей в том порядке, в котором они перечислены в пункте CRAB.

3.8. Сборка и запуск расчетного варианта

Центральным звеном системы Сафра является операция EXECUTE, предназначенная для сборки из хранящихся в архиве модулей конкретного расчетного варианта программы и передачи его на выполнение. Напомним, что для определения расчетного варианта необходимо задать соответствия между функциональными именами гнезд каркаса программы и архивными именами модулей, которые требуется подставить на их место в данном расчете.

Проиллюстрируем схему формирования расчетного варианта на следующем примере (рис.3.1). Пусть на базе имеющегося программного фонда требуется построить программу,

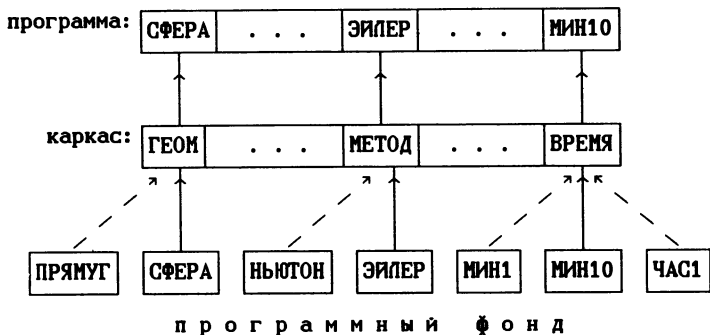


Рис.3.1. Формирование расчетного варианта программы

осуществляющую десятиминутный расчет по методу Эйлера в сферической геометрии. Иначе говоря, в каркасе программы имеются гнезда, отвечающие за метод расчета (РАСЧЕТ), геометрию (ГЕОМ), в которой производится расчет, и за время проведения расчета (ВРЕМЯ). Для каждого из этих гнезд имеется несколько реализаций, из которых надлежит выбрать компоненты формируемой программы в соответствии с требованиями задачи. В данном случае это можно сделать с помощью пункта EXECUTE, в теле которого содержатся указания о заполнении гнезд каркаса:

```
+EXECUTE
MATCH  МЕТОД = ЭЙЛЕР
MATCH  ГЕОМ  = СФЕРА
MATCH  ВРЕМЯ = МИН10
```

На основе этой информации Сафра сформирует (см.рис.3.1) и запустит на счет необходимый вариант программы.

Рассмотрим теперь используемые здесь механизмы более подробно.

Предложение INSERT, уже упоминавшееся в п.3.1, является одним из основных средств конструирования модульной структуры при работе с системой Сафра. Приняв решение об оформлении некоторой части своей задачи в виде модуля типа вставка (MACRO), пользователь определяет функциональное имя этого модуля, создает одну или несколько реализаций данного модуля, записывает их в архив под уникальными архивными именами, и, наконец, вместо текста выделенного модуля в соответствующих ему местах программ

(или задания) записывает предложение INSERT. Предложение INSERT имеет вид

<признак> INSERT <функциональное имя>

где **<признак>** — признак управляющего предложения; **<функциональное имя>** — имя, присвоенное выделенному модулю (гнезду каркаса) при функциональной модуляризации, которое представляет собой идентификатор, состоящий не более чем из шести символов.

Предложения INSERT записываются в архив наряду с остальными предложениями текста модуля и воспроизводятся при последующих распечатках, но при сборке расчетного варианта они заменяются текстами архивных модулей. В пункте EXECUTE любому функциональному имени, записанному в предложении INSERT, может быть поставлено в соответствие некоторое архивное имя модуля, текст которого будет подставляться в этом случае на место предложения INSERT. Если же такое соответствие не задано, выполняется подстановка текста модуля с архивным именем, совпадающим с функциональным именем, записанным в предложении INSERT.

Приведем пример использования предложения INSERT. Пусть при выполнении функциональной модуляризации принято решение о выделении в качестве модуля COMMON-блока с именем COMMAR. В этом случае использующая указанный COMMON-блок программа SOLVE, имеющая вид

```
SUBROUTINE SOLVE
COMMON /COMMAR/
1 N, AMAX, AMIN
2 , RF(300), F(200)
PRINT 1 N, RF(1), RF(N)
. . . . .
END
```

записывается в архив следующим образом:

```
+STORE PRUNIT:НЬЮТОН
SUBROUTINE SOLVE
+INSERT COMMAR
PRINT 1 N, RF(1), RF(N)
. . . . .
END
```

В результате выполнения этого пункта STORE в архив запишется под именем НЬЮТОН подпрограмма SOLVE, в тексте

которой вместо предложений, соответствующих описанию COMMON-блока COMMAR записано предложение INSERT. В подобном же виде записываются в архив и остальные использующие COMMAR модули. Текст описания COMMON-блока записывается в архив в виде модуля типа вставка (MACRO):

```
+STORE COMMAR
COMMON /COMMAR/
1 N, AMAX, AMIN
2 , RF(300), F(200)
```

Результатом такой организации использования COMMAR, помимо экономии записи, является существенное облегчение внесения изменений в описание блока. Дело в том, что подстановка текста модуля COMMAR производится только в момент сборки расчетного варианта. Поэтому внесение изменения в текст COMMAR изменяет фактически все тексты использующих его модулей, компенсируя тем самым известную слабость языка Фортран.

Кроме того, допустимо сосуществование в архиве нескольких различных вариантов блока COMMAR. Каждый вариант должен быть записан в архив под своим уникальным именем, например

```
+STORE NEWCOM
COMMON /COMMAR/
1 N, AMAX, AMIN
2 , RF(300), F(200)
3 .TAU
```

При составлении расчетного варианта задачи можно указать (посредством описываемого ниже MATCH-предложения), какой именно вариант реализации блока COMMAR следует использовать. Таким образом, имеется возможность накапливать в архиве различные реализации отдельных частей программ, методов расчета и т.д. и оперативно компоновать из этих частей посредством операции EXECUTE готовые расчетные программы

Пункт EXECUTE имеет заголовок следующего вида

<признак> EXECUTE

Тело пункта может содержать MATCH-предложения вида

MATCH <функциональное имя> = <архивное имя>

где <функциональное имя> – имя гнезда каркаса собираемой программы; <архивное имя> – архивное имя модуля, реализующего соответствующую функциональную часть в данном расчете.

Кроме MATCH-предложений тело пункта EXECUTE может содержать ссылки на модули типа вариант (VARIANT), записываемые в форме предложений INSEKт. Модуль типа вариант представляет собой группу MATCH-предложений, а ссылка на него в пункте EXECUTE означает, что вся эта группа будет участвовать в формировании варианта.

Если сборка расчетного варианта заканчивается успешно, на листинге задания печатается соответствующее сообщение, например

+++++ СФОРМИРОВАНА ЗАДАЧА ШИФР 031101000000

Перейдем теперь к описанию механизма сборки расчетного варианта. Расчетные задачи, формируемые Сафрой, работают на ЭВМ БЭСМ-6 в рамках операционной системы Диспак и мониторной системы Дубна. Это обстоятельство предопределило принятие соглашения, в соответствии с которым любой расчетный вариант составляется из двух функциональных частей: ДИСПАК – паспорт задачи для операционной системы, ДУБНА – задание для мониторной системы Дубна. Другими словами, текст любой расчетной задачи представляет собой результат расширения конструкции вида

+INSERT ДИСПАК
+INSERT ДУБНА

Рассмотрим сначала формальный механизм генерации текста, а затем перейдем к описанию содержания каждой из упомянутых частей. В основе механизма генерации текста расчетной задачи лежит следующий алгоритм расширения предложения INSERT.

Шаг 1. Определяется архивное имя, соответствующее функциональному имени, записанному в предложении INSERT. Если в теле пункта EXECUTE (с учетом всех упомянутых в нем модулей типа вариант) отсутствуют MATCH-предложения с данным функциональным именем в левой части, в качестве архивного имени берется функциональное имя. Если такие MATCH-предложения есть, среди них выбирается последнее и в качестве архивного имени берется архивное имя, записанное в его правой части.

Шаг 2. Предложения текста модуля с определенным на шаге 1 архивным именем просматриваются одно за другим и обрабатываются следующим образом:

1) С-предложения (комментарии – см.п.3.6) игнорируются;

2) предложения INSERT расширяются согласно описываемому алгоритму;

3) из S-предложений (см.п.3.7) в генерируемый текст задачи записывается только содержащийся в них <произвольный текст>;

4) остальные предложения переписываются в генерируемый текст без изменений.

Теперь можно определить, каким образом собирается расчетный вариант, состоящий, как уже упоминалось, из функциональных частей ДИСПАК и ДУБНА.

Часть ДИСПАК имеет обычно довольно простую структуру, поскольку включает в себя только разделы паспорта ОС Диспак. К ним добавляется (системой) заказ магнитного носителя, на котором расположен архив Сафры.

Часть Дубна представляет собой задание для мониторной системы Дубна. С помощью пункта EXECUTE, вообще говоря, можно сформировать часть ДУБНА произвольного вида. Однако если в формируемой задаче предполагается использовать механизм автоматизации процессов трансляции, то задание для мониторной системы должно иметь вид

***NAME ...**

.....

Ссылки на программные единицы, входящие в каркас.

***PERSO:62060,CONT**

Вызов программы связи

***CALL TEMPO**

с архивом Сафры.

.....

Предложения ***CALL FICMEMORY,**

***EXECUTE**

***MAIN ...** и др.

.....

Начальные данные расчета.

***END FILE**

В результате выполнения такого задания сначала во временную библиотеку мониторной системы записываются объектные модули, соответствующие всем участвующим в расчете программным единицам, а затем производятся сборка и выполнение программы.

Отметим, что текст, соответствующий исходным данным, используемым в расчетной задаче, также формируется из текстов архивных модулей. Как и любая другая часть задания, исходные данные могут обладать достаточно сложной организацией, основанной на использовании предложе-

ний INSERT. В этом случае к ним применим весь аппарат функциональных и архивных имен

Программные единицы (PRUNIT). Подобные схемы задания для мониторной системы обычно используются при решении вычислительных задач. Основная особенность работы по приведенной схеме в системе Сафры заключается в том, что, хотя между предложениями *NAME и *PERSO:62060,CONT должны быть как или иначе упомянуты все используемые в данном расчете программные единицы, транслироваться будут лишь те из них, которые не транслировались ни в одном из выполнявшихся ранее заданий или же подвергались редактированию после трансляции. Тексты транслировавшихся ранее программных единиц при генерации части ДУБНА исключаются из задания, а из архива Сафры выбираются соответствующие им объектные модули и с помощью программы ТЕМПО, вызываемой предложением *CALL TEMPO, черпываются во временную библиотеку мониторной системы. Эта же программа записывает в архив Сафры объектные модули, соответствующие транслируемым в данном задании программным единицам

Исключение из задания для мониторной системы текстов транслировавшихся ранее программных единиц происходит перед шагом 2 описанного выше алгоритма расширения предложений INSERT. Для каждого архивного модуля, участвующего в генерации задачи, прежде всего проверяется его тип. Если тип модуля — допускающая автономную трансляцию программная единица (PRUNIT), то сначала в архиве Сафры ищется соответствующий ей объектный модуль. Если модуль найден, предложению INSERT, содержащему функциональное имя программной единицы, будет соответствовать пустое расширение, а в формируемую задачу будет передано сообщение о необходимости дозаписи из архива Сафры во временную библиотеку Дубны соответствующего объектного модуля.

Итак, пользователь, желающий поручить Сафре хлопоты, связанные с хранением и использованием объектных модулей, должен все входящие в его расчеты программные единицы оформлять в виде модулей типа PRUNIT. Такие модули должны представлять собой готовые фрагменты задания для системы Дубна, т.е. в них необходимо предусмотреть предложения, задающие в соответствии с принятыми в мониторной системе соглашениями транслятор, применяемый к данной программной единице, и, возможно, некоторую дополнительную информацию. Все оформленные таким образом программные единицы должны быть включены в часть ДУБНА

посредством предложений INSERT, причем записанное в таком предложении функциональное имя должно совпадать с именем подпрограммы, хранящейся в подставляемом на место INSERT архивном модуле.

Задачи, порождаемые в результате выполнения пункта EXECUTE, снабжаются «шапкой», в которой печатаются все соответствия вида <функциональное имя> = <архивное имя>, заданные в теле пункта, и список функциональных имен транслируемых программных единиц. Кроме того, распечатываются в алфавитном порядке архивные имена всех модулей, участвующих в расчете. Листинг, выдаваемый вслед за шапкой, представляет собой обычную распечатку мониторной системы Дубна.

Модули типа вариант (VARIANT). В расчетах, выполняемых с помощью системы Сафра, используется обычно от 50 до 150 модулей. Явное выписывание всех необходимых для них соответствий между функциональными и архивными именами в теле формирующего расчетную задачу пункта EXECUTE довольно обременительно. Облегчить формирование варианта, сократив объем задаваемой в пункте EXECUTE информации, позволяет использование модулей типа вариант (VARIANT).

Дело в том, что обычно от расчета к расчету набор соответствий меняется лишь незначительно. В этом случае можно выделить общую часть соответствий, оформить ее в виде модуля типа вариант, записать этот модуль в архив Сафры и в дальнейшем ссылаться на него в пунктах EXECUTE. Тогда в теле пункта EXECUTE явно будет записываться лишь необходимый минимум пар.

Ссылка на модуль типа вариант оформляется в пункте EXECUTE в виде предложения INSERT. Текст модуля типа вариант, помимо явных указаний соответствий между именами посредством MATCH-предложений, может также содержать записанные в виде предложений INSERT ссылки на вложенные модули этого типа. Интерпретация пункта EXECUTE начинается с расширения всех входящих в него предложений INSERT. Результатом расширения является последовательность MATCH-предложений, которая и используется затем при формировании расчетного варианта.

Альтернативные схемы сборки. Рассмотренная выше схема сборки Сафры-БЭСМ-6 в некоторых случаях причиняет程序员 известное неудобство. Пусть, например, в некоторое гнездо каркаса подставляется фрагмент алгоритма, содержащий вызов автономно транслируемой подпрограммы. Это означает, что в части ДУБНА должно

содержаться предложение INSERT с функциональным именем этой подпрограммы. Прикладной программист, вообще говоря, обязан выяснить, присутствует ли в части ДУБНА такое предложение, и если нет, то включить его туда.

Между тем широко известна схема сборки программы, избавляющая программиста от явного перечисления всех входящих в нее автономно транслируемых частей. При этой схеме, которую мы назовем *корневой*, задается только головной модуль программы, т.е. корень ее дерева вызовов. В процессе сборки выявляются все внешние связи головного модуля, и к формируемой программе добавляются из библиотеки модули, «материализующие» эти связи. Корневая сборка заканчивается, когда все внешние связи формируемой программы конкретизированы.

Каркасный подход к конструированию программ, используемый в системе Сафра, может столь же успешно базироваться и на корневой схеме сборки. Для этого достаточно в процессе конкретизации внешних связей каждый раз осуществлять переход от функционального имени, фигурирующего в качестве внешнего в вызывающем модуле, к архивному имени, определяющему ту из нескольких возможных реализаций, которая должна быть включена в собираемую программу.

Тут, по-видимому, содержится и ответ на вопрос, почему и разработчики штатного системного обеспечения, и прикладные программисты нередко отказываются от корневой схемы сборки, несмотря на ее очевидные достоинства. Он заключается в том, что традиционная реализация корневой схемы не допускает вмешательства в процесс сборки, не позволяя тем самым осуществлять гибкое управление конфигурацией программы. В Сафре искомая гибкость корневой схемы может быть достигнута за счет регулярного перехода в процессе сборки от функциональных к архивным именам.

Применить корневую сборку в Сафре-БЭСМ-6 не удалось из-за сложностей в организации управляемой серии трансляций в мониторинной системе Дубна, однако уже в Сафре-ЕС эта схема сборки была успешно реализована. С точки зрения реализации корневая сборка несколько более трудоемка, так как здесь требуется детальное изучение структуры объектного модуля.

Следует отметить, что в Сафре корневая схема накладывает некоторые ограничения на участвующие в сборке модули. Если вызываемый автономно транслируемый модуль имеет несколько входов и вызываемому модулю требуется не основной (определяемый функциональным именем), а побоч-

ный вход, то обращение к побочному входу должно быть предварено упоминанием основного. Дело в том, что Сафра не предполагает, что к моменту начала сборки все участвующие в ней модули оттранслированы (это повлекло бы за собой выполнение избыточной работы по перетрансляции). Поэтому при сборке (точнее, при поиске вызываемого модуля) можно опираться только на явно заданные атрибуты текстов модулей, а задание побочных входов в качестве атрибутов весьма нежелательно. В то же время необходимое здесь предварительное упоминание основного входа является обязательным требованием большинства используемых в рамках Сафры алгоритмических языков, и поэтому пользователями Сафры рассмотренное ограничение корневой сборки практически не ощущается.

Говоря об альтернативных схемах сборки, необходимо упомянуть также, что размер задания, описывающего формируемый вариант программы, можно было бы вдвое сократить, указывая в нем вместо пар <функциональное имя> = <архивное имя> только <архивное имя>. Нетрудно видеть, что для этого достаточно сделать функциональное имя обязательным атрибутом каждого хранящегося в архиве Сафры модуля. Такое решение могло бы быть реализовано и в Сафре-БЭСМ-6, и в Сафре-ЕС, однако ни в том, ни в другом случае этого сделано не было, поскольку задание пар представляется более наглядным.

Наконец, можно поставить под сомнение одно из краеугольных предположений, положенных в основание Сафры. Почему вновь появляющиеся в ходе вычислительного эксперимента модули всегда заменяют существующие части расчетных программ, а не дополняют их? Например, при исследовании некоторой установки, состоящей из четырех узлов (каждому из которых соответствует моделирующий его поведение фрагмент алгоритма) может быть принято решение добавить еще один, пятый узел. Конечно же, модель пятого узла дополнит программу, а не заменит собой какую-либо ее часть.

И все же для тех классов задач, к решению которых привлекалась Сафра, такого рода изменения программ требовались сравнительно редко, и для них не были реализованы соответствующие системные средства. В то же время включение подобных средств позволило бы существенно обогатить Сафру, превратив ее в систему поддержки любых развивающихся программ. О том, как могли бы выглядеть средства программирования «вширь», можно прочитать, например, в [116,117].

3.9. Вспомогательные операции

Изменение признака (FLAG). Признак управляющего предложения предназначен для выделения управляющих предложений языка заданий пакета. Введение признака обусловлено тем, что в тексте задания наряду с управляющими предложениями могут встречаться, например, написанные на некотором алгоритмическом языке фрагменты программ, предназначенные для записи в архив. Необходимо иметь возможность просто и эффективно распознавать управляющие предложения без наложения ограничений на используемые языки.

В качестве признака управляющего предложения должна выбираться такая комбинация из символов «+», «-», «*», «/» (не более трех), которая не встречается в первых позициях предложений фрагментов программ, содержащихся в этом задании. Удобнее всего пользоваться признаком, который не может появиться в предложениях программ из-за синтаксических особенностей языка программирования. Например, в первой позиции предложения Фортрана не может встретиться символ «+», поэтому он обычно используется в качестве признака в заданиях, содержащих фрагменты программ на Фортране.

Если же в одном задании нужно записать части программ на различных языках, для которых по тем или иным причинам используются различные признаки, можно воспользоваться пунктом FLAG вида

<признак> FLAG

Тело пункта FLAG пусто. В качестве признака в пункте FLAG записывается признак, использовавшийся в непосредственно предшествующих ему пунктах задания. Пункты, следующие за пунктом FLAG, могут иметь другой признак, допускаемый синтаксисом языка заданий. В задании может содержаться произвольное число пунктов FLAG. Изменение признака после пункта FLAG не обязательно, можно пользоваться старым значением. Следует иметь в виду, что в соответствии с синтаксисом языка изменить признак внутри тела какого-либо пункта нельзя.

Приведем фрагмент протокола задания с использованием пункта FLAG:

```
+C СНАЧАЛА ПРИЗНАКОМ ЯВЛЯЕТСЯ СИМВОЛ "+"  
+DELETE DUMMY  
+++++ МОДУЛЬ DUMMY УНИЧТОЖЕН
```

```

+FLAG
+++++ ИЗМЕНЕН ПРИЗНАК: СТАРЫЙ=+  НОВЫЙ=-/
      -/С ТЕПЕРЬ ПРИЗНАК "-/"
      -/STORE РАСЧЕТ
      . . . . .
+++++ НОВЫЙ МОДУЛЬ РАСЧЕТ СОЗДАН
      -/FLAG
      -/С ПРИЗНАК НЕ ИЗМЕНИЛСЯ
      -/Е

```

Управление печатью протокола (LIST). Задания для системы Сафра обычно достаточно компактны, поэтому, как правило, нет необходимости в управлении листингом для уменьшения его размеров. Единственным исключением является случай, когда в задание включаются пункты STORE, предназначенные для записи в архив длинных текстов модулей. Распечатка текстов записываемых модулей полезна, если в дальнейшем предполагается подвергнуть их пакетному редактированию с помощью пункта EDIT (предложения тела пункта STORE нумеруются, и именно в терминах этих номеров выполняется впоследствии редактирование). Однако при использовании диалогового редактора нет необходимости в распечатке текстов тел пунктов STORE, и ее можно отключить с помощью пункта LIST.

Заголовок пункта LIST имеет вид

<признак> LIST <режим>

где <режим> принимает значения ON или OFF. Тело пункта LIST пусто. Выполнение пункта с параметром OFF задает отключение режима распечатки на листинге тел последующих пунктов STORE, с параметром ON включение режима распечатки. В начале задания считается включенным режим ON. При выключенном режиме распечатки на листинге на месте пункта STORE печатается лишь его заголовок и сообщение о результате выполнения. Отметим, что печать листинга для пунктов, отличных от STORE, не зависит от режима LIST.

3.10. Документирование

Важнейшей составной частью любой крупной программной системы является документация. Если система развивающаяся, то особую актуальность приобретает задача оперативной корректировки документации в соответствии с вносимыми в систему изменениями. Такое традиционное средство

документирования, как выпуск печатных руководств, становится в данном случае неудобным, поскольку руководства, как правило, не отражают текущего состояния системы в интервале между публикациями и часто успевают устареть еще в процессе их подготовки. Тем не менее потребность в документах, отражающих текущее состояние разработки, очевидна: именно такой документ требуется, в частности, при передаче системы во внешние организации.

Все высказанные выше соображения в полной мере могут быть отнесены к системе Сафра. Как сама система, так и функциональное наполнение создаваемых на ее основе пакетов интенсивно развиваются, и, следовательно, подход, связанный с публикацией печатных документов, не может полностью удовлетворить нужды разработчиков. Основная роль в данном случае отводится оперативным средствам документирования, которые позволяют быстро получать печатные документы, отражающие текущее состояние пакета, и предоставляют возможность простого и легкого внесения изменений в тексты существующих документов.

Создание средств документирования в Сафре существенно упростилось благодаря тому, что к моменту начала работы в составе программного обеспечения ЭВМ БЭСМ-6 уже существовала система АСПИД – *автоматизированная система подготовки публикаций и документов*. АСПИД предоставляет набор команд, позволяющих задавать требуемое расположение текста публикации на страницах.

Средства документирования Сафры разрабатывались с учетом совместимости с набором команд АСПИД, что позволило, в частности, автоматизировать подготовку документов к публикации. Однако требованию оперативного получения документов АСПИД отвечал не в полной мере. Хотя в системе и была предусмотрена выдача документов на АЦПУ, но назначение такой выдачи было несколько иное – выявление ошибок в исходном тексте и отладка расположения текста на страницах. Поэтому документ в АСПИДе выдавался в одностраничном формате, мало пригодном для последующего использования его в качестве справочного материала.

В то же время распечатка текущей документации на АЦПУ обладает требуемыми свойствами простоты и оперативности. Необходимо лишь организовать печать документов в удобном для последующего использования формате – на двух страницах. Эту возможность предоставляет пункт BOOK, заголовок которого имеет вид

<признак> **BOOK** <список архивных имен>

где <список архивных имен> содержит имена распечатываемых модулей. Список может быть продолжен с помощью предложений **MODULE**.

В результате выполнения пункта **BOOK** формируется задача, которая распечатывает на АЦПУ в двухстраничном формате с редактированием тексты модулей в том порядке, в котором они перечислены в пункте **BOOK**. Каждый модуль печатается с новой страницы. Распечатываемый текст выравнивается по правому краю страницы, при необходимости выполняется перенос слов по правилам орфографии, страницы нумеруются и т.д. Подробные правила оформления текстов документов можно найти в работе [108].

3.11. Операции над архивом

Запуск системы в ОС Диспак. Для работы с системой Сафра необходимо иметь записанными на некоторых внешних носителях системные программы (в дальнейшем называемые *системой*) и архив Сафры. Система и архив могут быть расположены в произвольных местах одного или различных носителей. Средства указания расположения архива рассматриваются ниже. Расположение системы задается соответствующими управляющими картами ОС Диспак и МС Дубна. Указание о расположении архива (**ARCHIVE**) необходимо лишь при нестандартном его расположении. Если же архив располагается стандартно, т.е. непосредственно вслед за системой, указывать его расположение не нужно.

*Задание текущего архива (**ARCHIVE**).* Операция **ARCHIVE** используется для указания расположения архива, а также для переключения в рамках одного задания с архива на архив. Заголовок пункта **ARCHIVE** имеет вид

<признак> **ARCHIVE** <том>: <зона>

где <том> – номер архивного тома, <зона> – номер зоны, начиная с которой расположен архив. Тело пункта **ARCHIVE** пусто.

Например, выполнение пункта

+ARCHIVE 722:0400

приведет к тому, что последующие операции задания будут выполняться над архивом, расположенным на магнитной ленте 722 начиная с 400-й зоны.

Если архив размещается не непосредственно вслед за системой, то пункт ARCHIVE должен быть первым в задании. Пункт ARCHIVE, используемый для переключения на другой архив, может быть расположен в любом месте задания. После его выполнения текущим становится указанный в нем архив.

Копирование архива (COPY) – сравнительно часто выполняемая операция. Она служит как для тиражирования содержимого архива, так и для обеспечения надежности хранения информации в архиве. Хотя специализированный архив Сафры поддерживает целостность информации при возникновении сбоев ЭВМ, однако если магнитные носители подвергаются механическим повреждениям, которые на БЭСМ-6 не так уж редки, то единственным источником восстановления информации становится чтение ее с копии.

В качестве параметров операции COPY указывается место, начиная с которого записывается копия архива. Как текущий архив, так и его копия могут располагаться в нескольких томах, номера которых записываются в теле пункта COPY. Архив, расположенный на нескольких томах, может использоваться только в операции COPY. Заголовок пункта COPY имеет вид

`<признак> COPY <том>: <зона>`

где <том> и <зона> указывают место, начиная с которого записывается копия архива. Если исходный архив или формируемая копия располагаются на нескольких томах, номера этих томов перечисляются в теле пункта в предложениях FROM (для исходного архива) и TO (для копии)

Например, в результате выполнения пункта

`+COPY 1766:0140
TO 1767`

начало исходного архива будет записано на ленту 1766 в зоны 140–777, а хвост его на ленту 1767 с нулевой зоны

Если копирование прошло успешно, на листинге печатается соответствующее сообщение, например

`+++++ СКОПИРОВАНО 600 БЛОКОВ`

Заметим, что по операции COPY производится копирование только фактически занятых архивом блоков и поэтому оно происходит быстрее, чем сплошное копирование через другие системы.

Изменение размера архива (SIZE) При образовании архива посредством операции CREATE (см. ниже) размер его устанавливается равным 40 зонам. Это означает, что, если объем хранимой в архиве информации превысит 40 зон, новая информация записана не будет и появится распечатка соответствующего диагностического сообщения.

Если все же требуется разместить в архиве информацию, занимающую свыше 40 зон, размер архива можно увеличить с помощью пункта SIZE вида

«признак» SIZE «размер»

где «размер» - восьмеричное число, кратное 40, задающее новый размер архива в зонах (блоках). Тело пункта SIZE пусто.

Посредством операции SIZE можно и уменьшать размер архива, освобождая часть пространства на внешнем носителе. Уменьшение размера, естественно, возможно лишь в случае, когда хранимая информация может быть размещена в архиве нового размера.

Копирование системы (SYSTEM) Вообще говоря, предпочтительнее использовать систему (системные программы Сафры), записанную на системный диск Сафры. Однако иногда требуется скопировать систему на заданный внешний носитель. Это можно сделать посредством пункта SYSTEM вида

«признак» SYSTEM «том»:«зона»

где «том» и «зона» задают место, начиная с которого записывается копия системы. Тело пункта SYSTEM пусто.

Образование нового архива (CREATE) Образовать новый (пустой) архив на заданном месте внешнего носителя можно с помощью пункта CREATE вида

«признак» CREATE «том»:«зона»

где «том» и «зона» задают начало формируемого архива на внешнем носителе. При формировании задается длина архива, равная 40 зонам. Копирование системы по операции CREATE не производится.

Перепись модулей из другого архива (SELECT) Выборочная перепись модулей из заданного архива в текущий выполняется с помощью операции SELECT. Заголовок пункта SELECT имеет вид

«признак» SELECT «том»:«зона» «список архивных имен»

где <том> и <зона> задают начало архива, из которого производится выбор модулей, а <список архивных имен> и, возможно, предложения MODULE содержат имена модулей, переписываемых в текущий архив.

Например, в результате выполнения пункта

```
+SELECT 2145:1140  
MODULE OLM101 OLM102  
MODULE DIA306
```

в текущий архив переписутся три модуля, расположенных в архиве, начинающемся с 1140-й зоны диска 2145.

Если в текущем архиве уже есть модуль с указанным именем, то перепись данного модуля не производится и выдается диагностическое сообщение. Диагностическое сообщение не выдается только в случае, если тексты модулей идентичны.

Г л а в а 4

СИСТЕМА ПНФ И ПАКЕТ ЗАЩИТА

4.1. Конструирование программ из нестандартизированных модулей

Каркасный подход к конструированию программ, положенный в основу рассмотренной в предыдущей главе системы Сафра, предполагает, вообще говоря, что все участвующие в конструировании модули разрабатываются в рамках единого набора соглашений. Для успешного применения каркасного подхода такие соглашения должны быть выработаны еще на стадии модульного анализа предметной области, т.е. непосредственное программирование начинается только тогда, когда для каждого модуля уже четко определены все его интерфейсные параметры. Однако на практике нередко требуется организовать совместное выполнение нескольких программ, написанных независимыми группами разработчиков, использовавшими различные, непохожие друг на друга интерфейсные соглашения.

Такая ситуация возникла, в частности, при проведении расчетов, связанных с проектированием защиты атомных реакторов. Несколько организаций долгое время независимо занимались моделированием различных областей радиационной защиты реактора. В каждой из этих организаций были созданы моделирующие программы объемом порядка сотен тысяч операторов. С функциональной точки зрения накопленных программ было вполне достаточно для того, чтобы выполнять комплексные расчеты, охватывающие все области радиационной защиты. Но на пути проведения комплексных расчетов существовало одно серьезное препятствие – информационная рассогласованность имеющихся программ.

Интерфейсные соглашения этих программ отличались чрезвычайной пестротой. Для хранения значений одного и того же физического показателя использовались то один, то несколько массивов, интерфейсные данные различным образом размещались во внешней памяти, некоторые расчеты

велись на одномерных моделях, другие — на двумерных и т.д.

Конечно, не могло быть и речи о том, чтобы разработать новые единые интерфейсные соглашения и заново переписать в соответствии с ними все программы, — такая работа отняла бы долгие годы. Более того, требовалось, чтобы деятельность, связанная с организацией совместного выполнения программ, велась безболезненно с точки зрения продолжающихся производственных расчетов, предполагающих автономное использование каждой из программ. Другими словами, тексты существующих программ не должны были подвергаться каким-либо модификациям.

При таких ограничениях решение поставленной задачи организации комплексных расчетов — становится практически однозначным. Для каждого выходного параметра некоторого модуля, потребляемого другим модулем, следует написать программу-преобразователь, осуществляющую перевод значения этого параметра из формата модуля-источника в формат, требующийся модулю-получателю. Далее при формировании расчетной цепочки, обеспечивающей последовательное выполнение нескольких модулей, необходимо предусмотреть включение в эту цепочку надлежащих преобразователей значений выходных параметров модулей в форматы входных параметров.

На рис.4.1 изображен пример подобной цепочки, обеспечивающей совместное выполнение нестандартизированных модулей А, В и С с помощью преобразователей Т1, Т2 и Т3

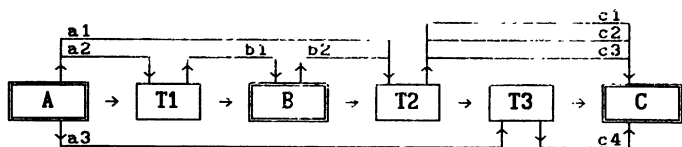


Рис.4.1. Цепочка модулей (А,В,С) с преобразователями

Здесь все выходные/входные параметры модулей проходят через преобразователи, причем преобразователь Т2 два выходных параметра (а1 модуля А и b2 модуля В) превращает в три входных параметра (с1, с2 и с3 модуля С)

Если бы для проведения реакторных расчетов требовалось составить только одну цепочку, подобную изображенной на рис.4.1, то создавать какие-либо системные средс-

тва для поддержки этой работы, разумеется, не было бы смысла. Однако, как показал предварительный анализ, число различных модульных цепочек, возникающих при решении задач реакторной защиты, составляет несколько десятков, причем в каждую цепочку включается обычно 5-6 функциональных модулей и несколько преобразователей. Хотя такие масштабы конструирования относительно невелики (напомним, что в системе Сафра реально конструируются десятки тысяч различных вариантов расчетных программ), тем не менее здесь уже вполне оправдано применение некоторого регулярного подхода, поддерживаемого соответствующими системными средствами.

Такой комплекс языковых и системных средств, обеспечивающих создание Пакетов прикладных программ на базе Нестандартизированных программных Фондов, предоставляет система ПНФ [118-120]. Эта система разработана в ИПМ имени М.В.Келдыша АН СССР для ЭВМ БЭСМ-6 в рамках операционной системы Диспак и мониторной системы Дубна.

4.2. Основные понятия системы ПНФ

Нестандартизированный программный фонд составляют программы, которые функционально полностью охватывают предметную область некоторого научно-технического приложения, но не могут быть непосредственно сопряжены между собой, так как они были написаны независимо, без учета их возможного совместного использования в рамках решения одной задачи. Элементы нестандартизированного фонда будем называть *квазимодулями* (в отличие от модулей, допускающих непосредственное сопряжение). В приложениях, связанных с решением больших комплексных задач, в частности, в задачах реакторной защиты, квазимодули характеризуются большими размерами, сложной логической структурой и значительной информационной рассогласованностью.

Система ПНФ предоставляет средства для превращения квазимодулей в модули, пригодные для совместной работы под управлением этой системы. Таким образом, использование системы ПНФ позволяет превратить нестандартизированный программный фонд в пакет прикладных программ, поддерживающий конструирование расчетных цепочек из модулей пакета. Рассмотрим технологию создания пакета прикладных программ с помощью системы ПНФ.

Работа по созданию пакета начинается с анализа расчетных контекстов, т.е. выявления всех возможных связей, устанавливаемых между модулями в различных цепочках. Для

всех входных и выходных параметров модулей, участвующих в таких связях, специфицируются их характеристики (размещение, структура и т.п.), значения которых используются в дальнейшем в качестве исходных данных для программ преобразователей.

Преобразователь извлекает значение выходного параметра модуля-источника из его области хранения, преобразует его в формат, требующийся модулю-потребителю, и записывает преобразованное значение в область хранения входного параметра модуля-потребителя. Любая встречающаяся в расчетных цепочках связь модулей по параметрам должна быть поддержана каким-либо преобразователем. Однако число преобразователей может быть существенно меньше числа различных связей модулей, поскольку один преобразователь может обеспечивать несколько сопряжений.

Область применимости преобразователя определяется форматами обрабатываемых им параметров модулей. Формат представляет собой перечень характеристик параметра с указанием диапазона допустимых значений для некоторых характеристик. Одним и тем же форматам могут отвечать несколько сопрягаемых модульных параметров, и тогда к любым из них применим преобразователь, специфицированный этими форматами.

Модуль в системе ПНФ формируется из заголовка, описывающего его интерфейс, и тела, задающего исходный квазимодуль. В заголовке модуля записываются значения характеристик его параметров. Состав описания параметра определяется форматом этого параметра: для каждой характеристики из числа составляющих формат необходимо указать ее значение для данного параметра. Например, чтобы указать, что описываемый параметр размещается в неименованном COMMON-блоке, необходимо в качестве значения характеристики РАЗМЕЩЕНИЕ задать слово COMMON (РАЗМЕЩЕНИЕ =COMMON).

Итак, в качестве входных данных преобразователь использует значения характеристик параметров, заданные в заголовках модулей. Входные и выходные данные преобразователя могут иметь как различные, так и совпадающие форматы. В последнем случае различие преобразуемых параметров обусловлено отличающимися значениями одних и тех же характеристик. Преобразователь может контролировать, имеют ли передаваемые ему данные надлежащие форматы.

Отметим, что преобразователь может, вообще говоря, осуществлять сопряжение для групп параметров (это необходимо, в частности, когда одному и тому же физическому

объекту в разных модулях соответствует различное число параметров).

Создание заголовков и преобразователей завершает формирование функционального и системного наполнений пакета. Теперь квазимодули, снабженные заголовками, превратились в полноценные модули, пригодные для конструирования из них расчетных цепочек. Единственное дополнительное усилие, которое требуется при таком конструировании, – вставление в цепочку между функциональными модулями необходимых преобразователей.

Заметим, что расстановку преобразователей в цепочке могла бы взять на себя и система ПНФ. Для этого пришлось бы немного дополнить информацию, задаваемую в описаниях модулей и преобразователей, а также использовать более мощный алгоритм компиляции расчетной цепочки. Однако оказалось, что развивать систему в данном направлении нецелесообразно, поскольку такая автоматизация не дает сколько-либо заметного выигрыша в производительности труда. Дело в том, что когда все необходимые преобразователи уже подготовлены, ручное конструирование цепочки даже при самом тщательном контроле занимает у прикладного программиста несколько минут. В то же время подготовка исходных данных для расчета по сконструированной цепочке требует нескольких часов.

Для описания заголовков модулей, свойств преобразователей, а также для формирования расчетных цепочек система ПНФ предоставляет необходимые языковые средства. В частности, задание расчетной цепочки выполняется на языке, являющемся расширением Фортрана, что позволяет включать в цепочку не только вызовы функциональных модулей и преобразователей, но и другие необходимые действия.

Оформленные надлежащим образом функциональные модули, преобразователи и расчетные цепочки хранятся в архиве пакета. При необходимости подключения к ведущимся расчетам нового квазимодуля потребуется пополнить архив пакета заголовком этого квазимодуля и, возможно, написать ряд новых преобразователей.

Перейдем теперь к рассмотрению отдельных языковых средств системы ПНФ. Следует иметь в виду, что приводимое ниже описание языка нельзя рассматривать как инструкцию, поскольку для упрощения изложения в нем изменены или опущены некоторые второстепенные конструкции.

4.3. Характеристики параметров модуля

Для каждого включаемого в пакет квазимодуля необходимо выявить и описать в его заголовке в качестве входных или выходных параметров следующие величины:

- входные и выходные файлы квазимодуля, расположенные на внешней памяти;
- скаляры и массивы, которые передаются от этого квазимодуля к другим или от других к этому и либо располагаются в COMMON-блоках, либо вводятся в качестве исходных данных задания

Синтаксис описания каждого из параметров модуля (квазимодуль с появлением заголовка превращается в модуль, и это позволяет нам употреблять термин «параметр модуля») имеет следующий вид.

```
«описание параметра модуля» := «имя параметра»  
                                ( «список характеристик» )  
«список характеристик» := «характеристика»  
                             [, «характеристика» ] ..  
«характеристика» := «имя характеристики»  
                    = «значение характеристики»  
                    [ ( «порожденный список характеристик» ) ]  
«порожденный список характеристик» :=  
    «список характеристик»  
«имя характеристики» := «слово»  
«значение характеристики» := «целое» | «слово» |  
                                ( «список целых» )  
«список целых» := «целое» [, «целое» ] ..
```

где «слово» - последовательность символов длиной не более 12, начинающаяся с буквы и состоящая из букв, цифр и символов «.» и «_»; «целое» - целая константа без знака

Таким образом, описание параметра модуля представляет собой список его характеристик. Описание записывается с первой позиции новой строки; при записи описания на нескольких строках признаком продолжения служит пробел в первой позиции.

В качестве объекта, для которого задаются характеристики, может выступать не только описываемый параметр модуля, но и значение некоторой его характеристики, являющееся в этом случае вспомогательным объектом, позволяющим задавать иерархию характеристик. Список характеристик, относящийся к вспомогательному объекту, будем называть порожденным списком.

Рассмотрим некоторые характеристики параметров, задаваемые в их описаниях. Конкретный набор характеристик, указываемых для каждого параметра модуля, определяется форматом параметра. Считается, что перечень характеристик, входящих в тот или иной формат, задается внешними по отношению к системе ПНФ средствами, т.е. вручную. Поэтому система не может проверить, все ли требуемые форматом характеристики параметра включены в заголовок модуля. Тем не менее при описании параметра рекомендуется указывать имя соответствующего ему формата, поскольку это позволяет впоследствии выявить такие грубые ошибки, как обращение к преобразователю с не предназначенным для него форматом входных данных. Характеристика ФОРМАТ записывается в виде

ФОРМАТ = <слово>

где <слово> задает имя формата

Характеристика ВИД указывает, является ли описываемый параметр входным (значение ВХОД), выходным (ВЫХОД) или модифицируемым (ВХОД-ВЫХОД).

Характеристика РАЗМЕЩЕНИЕ определяет место размещения параметра в оперативной памяти или на внешнем носителе.

РАЗМЕЩЕНИЕ=ОП означает, что описываемый параметр модуля принадлежит некоторому COMMON-блоку, в порожденном списке характеристик указывается имя этого COMMON-блока (имя не задается, если блок не помечен), длина COMMON-блока, длина параметра и, возможно, смещение относительно начала COMMON-блока

РАЗМЕЩЕНИЕ=ОП([ИМЯ-COMMON=<идентификатор>,
ДЛИНА-COMMON=<целое>,
[СМЕЩЕНИЕ=<целое>].]
ДЛИНА=<целое>)

РАЗМЕЩЕНИЕ=COMMON означает, что описываемый параметр занимает весь COMMON-блок:

РАЗМЕЩЕНИЕ=COMMON([ИМЯ-COMMON=<идентификатор>,
ДЛИНА=<целое>)

РАЗМЕЩЕНИЕ=ВП означает, что параметр размещается в указанном месте внешней памяти; координаты этого места задаются математическим номером устройства и номером зоны (тракта); в порожденном списке характеристик указываются математический номер устройства, смещение (в

зонах) от начала внешнего носителя и длина параметра в зонах:

**РАЗМЕЩЕНИЕ=ВП(НОМЕР=<целое>
[, СМЕЩЕНИЕ=<целое>]
[, ДЛИНА=<целое>])**

РАЗМЕЩЕНИЕ=КАНАЛ означает, что параметр располагается на внешней памяти, обращение к которой производится с указанием номера канала; в порожденном списке характеристик указываются номер канала и максимальное число зон, которое может занять описываемая величина:

**РАЗМЕЩЕНИЕ=КАНАЛ(НОМЕР=<целое>
[, ДЛИНА=<целое>])**

РАЗМЕЩЕНИЕ=ВВФАЙЛ означает, что параметр представляет собой вводной файл мониторной системы Дубна.

РАЗМЕЩЕНИЕ=ПК означает, что описываемый параметр располагается во вводном файле мониторной системы Дубна; в порожденном списке характеристик указываются номер записи в файле, смещение (номер позиции символа, по умолчанию – 0), длина параметра (количество элементов массива, по умолчанию – 1), формат Фортрана, по которому вводится элемент массива, и число позиций, занимаемое одним элементом массива:

**РАЗМЕЩЕНИЕ=ПК(НОМЕР=<целое>
[, СМЕЩЕНИЕ=<целое>]
[, ДЛИНА=<целое>]
, FORMAT=<слово>
, ЧПОЗ=<целое>)**

4.4. Описание модуля

Как уже упоминалось, в системе ПНФ каждый квазимодуль снабжается заголовком, содержащим описания всех входных и выходных параметров. В результате формируется функциональный модуль пакета, состоящий, таким образом, из заголовка, задающего интерфейс, и тела, задающего квазимодуль. Описание модуля на языке системы ПНФ имеет вид

**MODULE <имя модуля>
(MAIN=<имя главной программы>
/R <имя заголовка модуля>**

/R <имя тела модуля>
MODEND

В заголовке модуля содержатся описания всех параметров модуля:

HEADER <имя заголовка модуля>
<описание параметра модуля>
.....
HEAEND

Структура описаний параметров модуля рассматривалась в предыдущем разделе.

В теле модуля задаются сведения о программах, входящих в состав квазимодуля:

BODY <имя тела модуля>
[([**TRANSL**=<имя транслятора>,
 PERSO=<место>)]
[**/R** <имя программной единицы>]
.....
BODEND

(Заметим, что во второй и третьей строках записана металингвистическая формула, содержащая вложенные квадратные скобки «необязательности».)

Часть программ (точнее, программных единиц, допускающих автономную трансляцию) может быть задана в текстовом виде, и в этом случае их список задается с помощью предложений **/R**. Часть программ может быть взята из библиотеки (**PERSO**) объектных модулей мониторной системы Дубна. Расположение библиотеки на внешнем носителе задается параметром <место>, который имеет вид **NNZZZZ**, где **NN** – математический номер устройства, а **ZZZZ** – номер зоны.

Для программ, задаваемых в текстовом виде, следует указать <имя транслятора>, который должен быть к ним применен: **FORTRAN**, **FOREX** или **ALGOL**. Отметим, что <имя транслятора> может задаваться и при описании отдельных программных единиц. Поэтому в теле модуля указание о трансляторе записывается лишь в случае, если все программные единицы или большинство из них написаны на одном и том же языке программирования.

Если в теле модуля нет ссылок на программные единицы, то указание библиотеки объектных модулей должно быть

включено обязательно, в результате чего все программы будут извлекаться из этой библиотеки.

Отдельные программные единицы на языке системы ПНФ записываются следующим образом

```
UNIT <имя программной единицы>  
[(TRANSL=<имя транслятора>)]  
<текст программной единицы>  
UNIEND
```

В качестве примера приведем описание модуля ROZ6, участвующего в реальных производственных расчетах.

```
/C ОПИСАНИЕ МОДУЛЯ ROZ6  
MODULE ROZ6  
(MAIN=ROZ6M)  
/R ROZHDR  
/R ROZBOD  
MODEND  
  
/C ЗАГОЛОВОК МОДУЛЯ ROZ6  
HEADER ROZHDR  
DATA(ВИД=ВХОД, ФОРМАТ=ДАННЫЕ, РАЗМЕЩЕНИЕ=ВВФАЙЛ)  
CONROZ(ВИД=ВХОД, ФОРМАТ=КОНСТАНТЫ,  
    РАЗМЕЩЕНИЕ=КАНАЛ(НОМЕР=4))  
SOURC1(ВИД=ВХОД, ФОРМАТ=ИСТОЧНИК-Q1,  
    РАЗМЕЩЕНИЕ=КАНАЛ(НОМЕР=6), РАЗМЕЩЕНИЕ-Н.К.=ПК  
    (НОМЕР=1, СМЕЩЕНИЕ=15, ДЛИНА=1, FORMAT=I3, ЧПОЗ=3))  
SOURCG(ВИД=ВХОД-ВЫХОД, ФОРМАТ=ИСТОЧНИК-Q1,  
    РАЗМЕЩЕНИЕ=КАНАЛ(НОМЕР=7), РАЗМЕЩЕНИЕ-Н.К.=ПК  
    (НОМЕР=1, СМЕЩЕНИЕ=18, ДЛИНА=1, FORMAT=I3, ЧПОЗ=3))  
HEAEND  
  
/C ТЕЛО МОДУЛЯ ROZ6  
BODY ROZBOD  
(TRANSL=FOREX, PERSO=420020)  
/R ROZ6M  
BODEND  
  
UNIT ROZ6M  
    SUBROUTINE ROZ6M  
    . . . . .  
    END  
UNIEND
```

4.5. Преобразователь

Преобразователь обеспечивает передачу и преобразование группы P_1, \dots, P_m входных данных преобразователя (т.е. выходных параметров модулей) в группу P_{m+1}, \dots, P_n выходных данных преобразователя (т.е. входных параметров модулей). При построении расчетной цепочки в обращении к преобразователю указываются только эти параметры (P_1, \dots, P_n). Однако исходными данными для работы программы преобразователя служат не только значения выходных (P_1, \dots, P_m) параметров модулей, но и значения описанных в заголовках модулей характеристик всех (P_1, \dots, P_n) параметров, участвующих в обращении к преобразователю. Все необходимые преобразователю исходные данные указываются в заголовке преобразователя и передаются программе преобразователя через специальный COMMON-блок.

Преобразователь в целом и тело преобразователя описываются на языке системы ПНФ точно так же, как и модуль и его тело (см. п.4.4). Однако заголовок преобразователя не похож на заголовок модуля. Он имеет вид

```
HEADER <имя заголовка преобразователя>  
COND <условие применимости преобразователя>  
INITIAL <список исходных данных>  
ASSIGN <список присваиваний>  
HEAEND
```

Здесь <условие применимости преобразователя> представляет собой логическое выражение, истинность которого означает, что преобразователь применим к заданной совокупности параметров модулей; <список исходных данных> и <список присваиваний> определяют данные, передаваемые программе преобразователя через специальный COMMON-блок.

Логическое выражение в условии применимости состоит из отношений, связанных между собой логическими операциями («AND», «OR» и «NOT») с использованием круглых скобок. Например, логическое выражение

```
ФОРМАТ(1) .EQ. ИСТОЧНИК-Q1 .AND.  
ФОРМАТ(2) .EQ. ИСТОЧНИК-Q2
```

определяет, что форматом первого (в списке P_1, \dots, P_n) параметра преобразователя может быть только формат ИСТОЧНИК-Q1, а второго – ИСТОЧНИК-Q2.

Синтаксис отношения имеет вид

```
<отношение> ::= <операнд> .EQ. <операнд>
<операнд> ::= <атрибут> | <слово> | <целое>
<атрибут> ::= <имя характеристики>
               [<объект>](<параметр>)
<параметр> ::= <целое>
```

Здесь <атрибут> задает значение характеристики с именем <имя характеристики> для модульного параметра, который имеет номер <параметр> в списке P_1, \dots, P_n параметров преобразователя. Если <объект> в атрибуте опущен, то объектом является сам модульный параметр, иначе – вспомогательный объект (см. п.4.3), введенный в заголовке модуля при описании этого параметра.

Вслед за условием применимости в заголовке преобразователя записываются список имен исходных данных для программы преобразователя и предложение ASSIGN, в котором этим данным присваиваются значения.

В списке исходных данных через запятую перечисляются имена исходных данных в том порядке, в каком они располагаются в COMMON-блоке, используемом программой преобразователя. Если среди исходных данных встречаются массивы, то указывается их длина, т.е. элемент списка исходных данных имеет вид

```
<имя элемента>[<длина массива>]
```

Синтаксис списка присваиваний, задаваемого в предложении ASSIGN, имеет вид

```
<список присваиваний> ::= <группа>[, <группа>]
<группа> ::= <простая группа> | <условная группа>
<простая группа> ::= <присваивание>
                     [, <присваивание>]...
<условная группа> ::= IF <логическое выражение>
                       THEN (<простая группа>)
                       [ELSE (<простая группа>)]
<присваивание> ::= <имя элемента>=<правая часть>
<правая часть> ::= <атрибут> | (<параметр>) |
                  <слово> | <целое>
```

Здесь <логическое выражение>, <атрибут> и <параметр> имеют тот же смысл, что и в описанном выше предложении COND, а <имя элемента> – имя элемента исходных данных из предложения INITIAL.

Присваивания атрибута, слова и целого выполняются очевидным образом. Появление в правой части присваивания конструкции (<параметр>) означает, что исходным данным присваивается значение модульного (выходного) параметра с номером <параметр> в списке P_1, \dots, P_m .

При выполнении условной группы вычисляется логическое выражение и в зависимости от его истинности выполняется та или иная группа присваиваний.

В качестве примера приведем описание одного из преобразователей, использующихся для реакторных расчетов.

/C ОПИСАНИЕ ПРЕОБРАЗОВАТЕЛЯ TRZ

MODULE TRZ

(MAIN=TRZM)

/R TRZHDR

/R TRZBOD

MODEND

/C ЗАГОЛОВОК ПРЕОБРАЗОВАТЕЛЯ

HEADER TRZHDR

COND ФОРМАТ(1) .EQ. ИСТОЧНИК-Q1 .AND.

ФОРМАТ(2) .EQ. ИСТОЧНИК-Q2

INITIAL NTVK2D, NOKS, NROZ6

ASSIGN

IF РАЗМЕЩЕНИЕ(1) .EQ. КАНАЛ THEN NRQZ6=НОМЕР КАНАЛ(1)

IF РАЗМЕЩЕНИЕ(2) .EQ. КАНАЛ THEN NTVK2D=НОМЕР КАНАЛ(2)

NOKS=4

HEAEND

/C ТЕЛО ПРЕОБРАЗОВАТЕЛЯ

BODY TRZBOD

(PERSO=370640)

BODEND

4.6. Фрагменты расчетных цепочек

Записав в архив разрабатываемого пакета оформленные на языке системы ПНФ описания функциональных модулей и преобразователей, пользователь системы, вообще говоря, мог бы начинать составление цепочек и проведение расчетов по ним. Однако оказывается, что для обеспечения простоты, удобства и надежности последующей работы полезен еще один подготовительный этап – составление фрагментов расчетных цепочек. Использование фрагментов расчетных цепочек позволяет вести производственные расчеты,

практически не задумываясь о технических проблемах сопряжения модулей.

Описание фрагмента цепочки имеет вид

CHAIN <имя цепочки>
<подпрограмма на языке сборки>
CHAINEND

Язык сборки представляет собой Фортран, дополненный описаниями интерфейсных переменных, операторами вызова функциональных модулей и операторами вызова преобразователей.

Интерфейсные переменные служат для организации связи между фрагментами цепочек. Указание интерфейсной величины в качестве параметра вызова модуля обеспечивает связь по этому параметру с другими фрагментами расчетных цепочек. Описания интерфейсных переменных записываются через запятую в предложении **INTERFACE**, причем каждое описание имеет вид

<имя интерфейсной переменной>
[=<имя параметра>].<имя модуля>

Интерфейсной переменной приписываются все характеристики параметра модуля, указанного в описании интерфейсной переменной; <имя параметра> модуля можно опустить, если оно совпадает с именем интерфейсной переменной.

Оператор вызова функционального модуля в языке сборки имеет следующий синтаксис:

<вызов модуля> ::= <тип вызова> <имя модуля>
[(<интерфейс>)]
<тип вызова> ::= **EXECS** | **EXECD** | **EXECM**
<интерфейс> ::= **IN**: <список сопряжений>,
OUT: <список сопряжений>
<список сопряжений> ::= <сопряжение>
[, <сопряжение>]...
<сопряжение> ::= <имя параметра>
[=<имя цепочечной переменной>]

Здесь <тип вызова> модуля задает один из трех возможных механизмов вызова. Во-первых, модуль может загружаться статически (**EXECS**), т.е. до начала работы цепочки, как в мониторинной системе Дубна загружаются подпрограммы, вызываемые оператором **CALL**. Во-вторых, модуль может загружаться динамически (**EXECD**), т.е. в момент обращения, как в мониторинной системе Дубна загружаются подпрог-

раммы, вызываемые оператором CALL LOADGO. И в третьих, модуль может вызываться как главная программа (EXECM) в смысле мониторинг системы Дубна.

Кроме того, в операторе вызова модуля задается интерфейс между модулем и цепочкой, т.е. определяется, как должны передаваться данные из цепочки на вход модуля и из модуля в цепочку. Для этого задаются сопряжения между параметрами модуля и цепочечными переменными (цепочечная переменная это либо обычная переменная Фортрана, либо интерфейсная переменная).

Сопряжение выходного (или входного) параметра модуля с цепочечной переменной означает передачу значения выходного параметра цепочечной переменной в конце работы модуля (или передачу значения цепочечной переменной входному параметру перед началом работы модуля). Сопряжения для входных параметров модуля записываются в списке IN, для выходных в списке OUT. Если имена параметра модуля и цепочечной переменной совпадают, то имя цепочечной величины в сопряжении можно опустить.

Третья, в последняя конструкция языка сборки, расширяющая Фортран, оператор вызова преобразователя. Объектами действия преобразователя являются интерфейсные переменные. Задача преобразователя сопоставляя характеристики интерфейсных переменных (унаследованные ими от параметров модулей), выполнить необходимые преобразования над значениями одной из сопрягаемых переменных и присвоить полученное значение другой переменной. В общем случае могут сопрягаться группы интерфейсных переменных. Оператор вызова преобразователя имеет вид

TRANS <имя преобразователя>
(<список интерфейсных переменных> =
<список интерфейсных переменных>)

где слева от знака равенства перечисляются через запятую интерфейсные переменные, служащие выходными данными преобразователя (входными параметрами модулей), а справа от знака равенства - входные данные преобразователя (выходные параметры модулей).

В качестве примера приведем описания фрагментов расчетных цепочек TRQ и ROZSI

CHAIN TRQ
SUBROUTINE TRQ
INTERFACE SOURCT.TVK2D, SOURCI.ROZSI

```

        TRANS TRZ(SOURCI=SOURCT)
        RETURN
    END
CHAEND

CHAIN ROZS1
    SUBROUTINE ROZS1
    INTERFACE
        *CONROZ.OXS, SOURCI.ROZ6, SOURCG.ROZ6
    INTEGER IDATA
    С ДАННЫЕ ДЛЯ РАСЧЕТА ПЕРВОЙ ЧАСТИ ЗАЩИТЫ - ROZSID
    IDATA='ROZSID'
    С РАСЧЕТ ПЕРВОЙ ЧАСТИ ЗАЩИТЫ
    EXECM ROZ6(IN:DATA=IDATA, CONROZ, SOURCI,
        *
        OUT:SOURCG)
    RETURN
    END
CHAEND

```

4.7. Проведение расчета

Подготовив фрагменты расчетных цепочек, можно приступить к проведению расчета, которое включает в себя следующие действия:

- описание расчетной цепочки;
- задание начальных данных;
- запуск на счет.

Расчетная цепочка представляет собой программу на расширенном Фортране, задающую серию вызовов фрагментов расчетной цепочки. Она должна начинаться с оператора PROGRAM, т.е. быть главной программой в смысле мониторинг системы Дубна. Описание расчетной цепочки имеет вид

```

CHAIN <имя цепочки>
    PROGRAM <имя цепочки>
    EXECCH <имя фрагмента цепочки>
    . . . . .
    END
CHAEND

```

где EXECCH - оператор вызова расчетной цепочки - еще один оператор, расширяющий Фортран.

Начальные данные к расчету состоят из наборов начальных данных для вызываемых расчетных модулей. Описание набора начальных данных имеет вид

DDSET <имя набора начальных данных>
<начальные данные для расчетного модуля>
DDSEND

Имя соответствующего набора начальных данных задается в заголовке модуля.

Прежде чем перейти к директиве запуска расчетной цепочки на выполнение, необходимо познакомиться со структурой задания для системы ПНФ. Задание для системы представляет собой последовательность, состоящую из описаний объектов (модулей, преобразователей, фрагментов, цепочек, данных и т.д.) и директив, выполняющих различные действия над архивом функционального наполнения, инициирующих расчеты и т.д.

С помощью описаний задаются объекты, которые используются для формирования расчетной цепочки только в данном задании. Если же какие-либо объекты должны применяться в нескольких заданиях, то они записываются в архив функционального наполнения посредством директивы **STORE** вида

DIRECT STORE
<описание объекта>

.....
DIREND

Для обслуживания архива функционального наполнения используются также директивы **DELETE** (удаление объектов), **PRINT** (распечатка текстов объектов), **CATALOG** (распечатка каталога архива) и **EDIT** (редактирование текстов объектов в пакетном режиме). Эти директивы по назначению и синтаксическому оформлению близки к соответствующим операциям рассмотренной в предыдущей главе системы Сафра, и поэтому здесь они подробно описываться не будут.

Для запуска на счет служит директива **RUN**, которая должна быть последней в задании. Директива **RUN** имеет вид

DIRECT RUN
CHAIN <имя цепочки>
<дополнительная информация>
DIREND

Описания всех составляющих цепочку модулей и данных, а также описание самой цепочки должны либо содержаться в текущем задании, либо присутствовать в архиве функционального наполнения. В качестве дополнительной информации

к директиве RUN можно указывать место для размещения результатов расчета. Существует специальная форма директивы RUN, позволяющая осуществлять рестарт цепочки с запомненной ранее контрольной точки

4.8. Пакет ЗАЩИТА

Первым пакетом, разработанным с помощью инструментария базовой системы ПНФ, явился пакет ЗАЩИТА [121-123], предназначенный для автоматизации вычислительных работ, проводимых при конструировании радиационной защиты реакторов. В его состав вошли программы, которые можно разделить на следующие три класса: КОНСТАНТЫ, ПОЛЕ, ОБРАБОТКА. Эти программы создавались в разное время в различных организациях и использовали отличающиеся друг от друга интерфейсные соглашения, т.е. представляли собой нестандартизированный программный фонд. Посредством системы ПНФ удалось организовать эффективную совместную работу этих программ

Программы класса КОНСТАНТЫ преобразуют данные библиотек групповых микроконстант в макроконстанты, необходимые для решения многогрупповых систем уравнений переноса нейтронов и гамма-квантов, а также в константы для расчета функционалов полей излучения. Подготовка макроконстант базируется на использовании пакета ОКС, описанного в п.6.2 настоящей книги

В класс ПОЛЕ входят программы расчета распределений потоков нейтронов и гамма-квантов в реакторе и защите. Из-за ограниченности оперативной и внешней памяти ЭВМ БЭСМ-6 выполнение программ класса ПОЛЕ организуется в виде цепочки расчетов отдельных элементов защиты, связанных между собой источниками излучения и граничными условиями. Формирование таких цепочек основывается на средствах, предоставляемых системой ПНФ, которая, таким образом, используется не только для объединения программ из различных классов, но и для организации совместной работы нескольких программ одного класса

Программы класса ПОЛЕ получают и/или вырабатывают в качестве параметров многочисленные информационные массивы, определяющие, в частности:

- геометрические и физические параметры рассчитываемой системы «реактор + защита», задающие форму и размеры геометрических зон, состав этих зон, их температуру и т.п.;

- групповые макроконстанты, полученные в результате работы программ класса КОНСТАНТЫ;
- распределение внутренних (объемных) и внешних (поверхностных) источников излучения;
- плотности потоков излучения;
- функционалы поля излучения;
- сведения о расчетном алгоритме.

Программы класса ОБРАБОТКА осуществляют различные преобразования полученных результатов (расчет функционалов, пересчет геометрических или физических параметров системы в задачах оптимизации и т.д.) и выдачу результатов на внешние устройства (АЦПУ, графопостроитель, дисплей) в удобной пользователю форме.

Для хранения полученных результатов в пакете ЗАЩИТА создан специальный архив результатов счета. Можно любому расчету присвоить имя (задаваемое в предложении VARIANT директивы RUN), а затем с помощью этого имени ссылаться на вычисленные в нем данные. Например, директива RUN вида

```
DIRECT RUN
CHAIN TVKROZ
VARIANT TVKRO4
VALUE SOURCT TRO1
DIREND
```

предписывает выполнить расчет цепочки TVKROZ, записывая в архив результаты расчета под именем TVKRO4 (точнее, выходные параметры записываются в архив под составными именами, включающими имя параметра и имя расчета), а в качестве исходных данных для параметра SOURCT взять его значение, полученное при выполнении расчета с именем TRO1.

Для работы с архивом результатов в язык заданий пакета ЗАЩИТА введены директивы, позволяющие распечатывать каталог архива и уничтожить либо все результаты указанного расчета, либо значения отдельных параметров.

Г л а в а 5

АНАЛИЗАТОР PLAN: ОБРАБОТКА ДАННЫХ ФИЗИЧЕСКИХ ЭКСПЕРИМЕНТОВ

5.1. Архитектура

Вопросы создания математических методов и программных средств для обработки и интерпретации данных физических экспериментов составляют одно из главных направлений в проблеме автоматизации научных исследований. В настоящей главе мы рассмотрим один из возможных подходов к организации системного обеспечения пакетов, относящихся к важному разделу этого весьма обширного направления [16].

В современном физическом эксперименте, где среда подвергается сверхсильным воздействиям и параметры вещества имеют экстремальные значения, прямое измерение основных характеристик (температуры, плотности, давления, скорости и др.) затруднено или вообще невозможно. Поэтому часто информацию о значениях основных физических параметров приходится извлекать путем математической обработки косвенных экспериментальных данных, получаемых в виде фотоснимков, осциллограмм, интерферограмм и т.п. Такого рода математическое обслуживание физического эксперимента наиболее эффективно осуществляется пакетами программ, реализуемыми в форме многоцелевых проблемно-ориентированных систем [124–126].

Существенные достоинства таких систем прежде всего выражаются в том, что они, во-первых, обеспечивают возможность сплошной математической обработки экспериментальных данных, т.е. проведение в достаточно унифицированной форме решения ряда смежных задач, возникающих при переработке результатов физического эксперимента, и, во-вторых, делают вычислительную систему квалифицированным помощником исследователя, не имеющего специальной математической и программистской подготовки. Типичными задачами, решаемыми в ходе сплошной математической обра-

ботки результатов наблюдений, являются следующие [16,17]:

- получение предварительно обработанных выходных данных эксперимента;
- определение наилучшей в выбранном классе модели экспериментальной установки;
- интерпретация результатов эксперимента в выбранном классе моделей исследуемого объекта;
- определение значений управляющих параметров установки, при которых характерные элементы изучаемых типичных структур получают наилучшее разрешение;
- численное моделирование, т.е. вычисление с помощью модели установки по типичным структурам (моделям) изучаемого объекта точных выходных данных «модельного эксперимента».

Следует отметить, что принципы и методы, определяющие логическую структуру и программную реализацию рассматриваемых систем обработки, носят достаточно общий характер и поэтому обеспечивают при соответствующем развитии функционального наполнения возможность использовать каждую из этих систем при обработке данных экспериментов других типов. В связи с этим термин «многоцелевая» отражает функциональную многоаспектность системы не только в смысле разнообразия задач, решаемых при обработке результатов эксперимента, но и в смысле широты класса обслуживаемых экспериментов. Вместе с тем системное наполнение каждого такого пакета уникально и не ориентировано на самостоятельное использование при создании другого пакета.

Возникает вопрос: нельзя ли выделить в структуре подобных пакетов общую часть, т.е. разработать системное обеспечение, предназначенное для создания, развития и эксплуатации широкого класса многоцелевых проблемно-ориентированных систем обработки экспериментальных данных?

С точки зрения постановки такой задачи прежде всего необходимо отметить, что специфика прикладной деятельности, для обслуживания которой предназначен пакет, всегда находит отражение в его структурных составляющих. Это проявляется и в виде зависимости состава модулей функционального наполнения от предметной области приложения, и в виде соответствия конструктивов языка заданий дисциплине организации и проведения расчетов, принятой в приложении. Специфика прикладной деятельности, естественно, учитывается и в архитектуре системного наполнения, поскольку именно оно, интерпретируя с помощью

модулей функционального наполнения инструкции языка заданий, реализует дисциплину работы пользователя с пакетом.

В связи с этим целесообразно начинать разработку системного обеспечения для класса пакетов, характеризующих общностью проблематики (в данном случае имеется в виду проблематика сплошной обработки данных физических экспериментов) с выделения базовой дисциплины работы с пакетом. При этом под базовой дисциплиной мы понимаем обстоятельства, правила и формы проведения вычислений, инвариантные относительно широкого круга предметных областей (т.е. типов физических экспериментов), рассматриваемых в данной проблематике. Далее, ориентируясь на базовую дисциплину, можно специфицировать связанные с ней инвариантные свойства системного обеспечения рассматриваемого класса пакетов. Так, для языка заданий можно установить общий стиль формулирования заданий и определить внешний синтаксис, т.е. представление языковых конструкций, не зависящее от конкретных типов решаемых задач. В свою очередь для системного наполнения можно определить основные принципы регламента модуляризации прикладных программ, включаемых в пакет, зафиксировать форму организации межмодульного интерфейса (по управлению и по данным), установить способ исполнения расчетной цепочки, выбрать режим взаимодействия пользователя с пакетом при исполнении заданий и схему внешнего информационного обслуживания.

Затем, руководствуясь полученными спецификациями, необходимо разработать (или адаптировать имеющийся) комплекс языковых и программных средств, который будет обеспечивать поддержку базовой дисциплины расчетов. На основе такого комплекса (будем называть его базовым системным обеспечением) можно уже создавать родственные в дисциплинарном отношении пакеты, отличающиеся друг от друга функциональными наполнениями (т.е. обслуживаемыми предметными областями) и языками заданий (с точки зрения их внутреннего синтаксиса и семантики).

При такой организации разработки класса пакетов, естественно, возникает вопрос о том, как осуществить переход от базового системного обеспечения к конкретному пакету (рис.5.1). Вообще говоря, такой переход можно выполнить, ограничившись только средствами штатного программного обеспечения (трансляторы, редакторы текстов и внешних связей, возможности файловой системы и т.д.). Правда, при этом от разработчиков пакетов потребуются

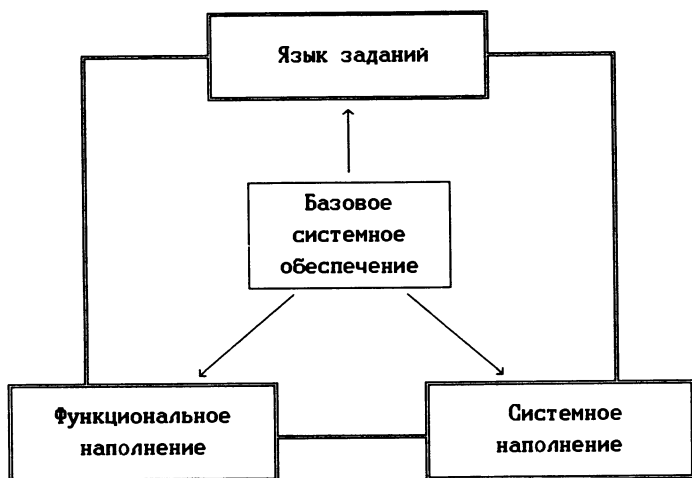


Рис 51 От базового обеспечения – к пакету

достаточно глубокие знания возможностей штатных системных программ и умение использовать их с учетом внутренних механизмов базового системного обеспечения. С другой стороны, процесс перехода от базового системного обеспечения к пакету может быть значительно упрощен, если предусмотреть специальные инструментальные средства, автоматизирующие основные этапы этого процесса.

В этом случае задачу создания универсального системного обеспечения для пакетов обработки данных физических экспериментов можно определить как задачу разработки соответствующей инструментально-базовой системы. Рассмотрим теперь основные особенности многоцелевых систем сплошной математической обработки экспериментальных данных, которые должны быть учтены в архитектуре инструментально-базовой системы.

Квалифицированное обслуживание пользователей, не имеющих специальной программистской подготовки (так называемых конечных пользователей), прежде всего, предполагает, что общение с системой осуществляется с помощью проблемно-ориентированного языка запросов [16,17]. Такой язык дает возможность составлять задание на работу системы в виде последовательности команд (приказов), сформулированных в терминах, которые либо непосредствен-

но заимствованы из сферы обработки научных наблюдений, либо естественно отражают основные задачи обработки данных, решаемые с помощью многоцелевой системы. Необходимо подчеркнуть, что функциональная многоаспектность системы обработки обуславливает два важных обстоятельства, касающихся организации общения пользователя с системой:

- во-первых, содержательное многообразие команд, применяемых при работе с системой; дело в том, что к услугам системы могут прибегать различные независимые группы конечных пользователей, каждая из которых занимается исследованием результатов экспериментов конкретного типа; в этих группах могут складываться свои специфические формы проведения расчетов, отражение которых пользователи хотят видеть в языковых средствах системы;

- во-вторых, постоянное расширение и модифицирование языка общения с системой вследствие развития функционального наполнения системы и увеличения числа использующих ее прикладных специалистов; другими словами, отмеченное выше содержательное многообразие команд системы не является фиксированным, а расширяется по мере развития и эксплуатации системы.

Если еще не накопились такие программные фонды, которые можно было бы использовать в качестве источников функциональных модулей при создании многоцелевых систем, то функциональное наполнение каждой многоцелевой системы в основном формируется из оригинальных программ, которые специально разработаны для данной системы в соответствии с принятым в ней регламентом программирования функциональных модулей.

Для многоцелевых систем характерно использование различных форм полуавтоматического планирования расчетных цепочек. Это объясняется тем, что для каждой типичной задачи (из числа решаемых в процессе сплошной математической обработки) можно заблаговременно определить один или несколько возможных вариантов оптимальной расчетной цепочки и указать четкие критерии выбора варианта.

Особенности операционных возможностей многоцелевых систем (жесткий интерфейс, исполнение задания в режиме интерпретации, диалог с пользователем) отражают их ориентацию на обеспечение широко используемого при автоматизации научных исследований человеко-машинного подхода к организации расчетов. Ориентация на такой подход обусловлена тем, что логическая структура процесса сплошной математической обработки, как правило, не может быть

полностью формализована заранее, поскольку она сложным образом зависит от значений исследуемых экспериментальных данных и получаемых в ходе обработки промежуточных результатов. Поэтому в большинстве случаев сеанс сплошной обработки данных наблюдений реализуется не по схеме сквозного выполнения задания, а по схеме его поэтапного выполнения, когда после завершения очередного этапа, описываемого одной или несколькими командами языка заданий, пользователь, оценив сложившуюся расчетную ситуацию и полученные результаты, имеет возможность указать, как должен протекать процесс обработки дальше.

Еще одной характерной чертой развитых многоцелевых систем обработки данных [105] является наличие в них средств внешнего информационного обслуживания, снабжающих пользователей сведениями о функциональных возможностях системы, о типах допустимых команд, их синтаксисе и семантике, а также обучающих правилам работы с системой.

Рассмотренные особенности пакетов обработки экспериментальных данных позволяют сделать вывод, что инструментально-базовая система, которая будет использоваться как универсальное системное обеспечение таких пакетов, должна удовлетворять следующим требованиям:

- язык системы должен содержать средства (будем в дальнейшем называть эти средства *языком генерации*), обеспечивающие возможность определения и расширения языка заданий;

- основными конструктивными единицами языка заданий должны быть пользовательские команды (запросы, приказы), характеризующиеся унифицированным внешним синтаксисом и выражаемые в терминах предметной области, охватываемой пакетом. Было бы удобно, например, чтобы язык генерации позволял определять семантику команды языка заданий путем описания возможных вариантов расчетной цепочки типичной задачи, которая будет решаться по данной команде, и указаний условий выбора того или иного из этих вариантов;

- необходимо, чтобы система устанавливала достаточно простой регламент внутренней модуляризации функционального наполнения и обеспечивала бы соответствующую программную поддержку этого регламента;

- необходимо, чтобы программные компоненты системы, являющиеся основой операционных средств создаваемого пакета, осуществляли бы эффективное (как по объему памяти, так и по времени процессора) исполнение пользовательских команд, а также обеспечивали бы возможность

обработки задания как в закрытом режиме, так и в режиме диалога с пользователем;

– необходимо, чтобы в системе был предусмотрен аппарат, позволяющий быстро реализовать внешнее информационное обслуживание (типа «меню») с конкретной предметной ориентацией.

Естественным архитектурным решением, удовлетворяющим перечисленным выше требованиям, является создание инструментально-базовой системы в форме анализатора проблемно-ориентированных языков, определяемых пользователем. В следующем разделе мы рассмотрим вопросы реализации и использования версии такого анализатора на машине БЭСМ-6.

5.2. Общая характеристика анализатора PLAN-БЭСМ-6

К моменту постановки задачи создания на машине БЭСМ-6 системного обеспечения для пакетов сплошной математической обработки экспериментальных данных уже была известна версия анализатора проблемных языков – PLAN (Problem Language ANalyzer), разработанная фирмой IBM [50]. Архитектура и внутренняя организация этого анализатора были подробно исследованы с точки зрения возможности использования его на машине БЭСМ-6 в качестве инструментально-базовой системы, ориентированной на создание пакетов обработки данных физических экспериментов.

Это исследование показало, что PLAN полностью удовлетворяет требованиям, которые предъявляются к системе такого рода, а его программная реализация характеризуется простой логической структурой и содержит ряд машинно-независимых составляющих. В связи с этим было решено разработать универсальное системное обеспечение для указанного класса пакетов в виде версии PLAN'а для машины БЭСМ-6. Такой подход, в частности, не только позволил существенно сократить сроки разработки, но и открыл благоприятные перспективы создания для машин типа БЭСМ-6 и ЕС ЭВМ семейства совместимых и мобильных пакетов.

Поскольку функциональные возможности штатного программного обеспечения машин БЭСМ-6 и ЕС ЭВМ существенно разнятся, реализация PLAN'а оказалась сопряжена с переработкой структуры и ряда управляющих модулей, в том числе монитора, процессора языка генерации и интерпретатора языка пользователя. Необходимо подчеркнуть, что при этом была обеспечена эквивалентность функций этих моду-

лей. Реализованная для машины БЭСМ-6 версия анализатора получила название PLAN-БЭСМ-6 [127,128].

Прежде чем перейти к обсуждению вопросов реализации анализатора PLAN-БЭСМ-6 и технологии его использования при создании пакетов для обработки данных физических экспериментов, рассмотрим кратко основные (проблемно и машинно независимые) понятия и принципы функционирования PLAN'a, которые отражают инструментально-базовый характер его архитектуры.

5.2.1. Базовые средства. Язык заданий пакета (называемый авторами PLAN'a языком пользователя) создается на базе установленных PLAN'ом формализмов, которые целиком переходят в каждый разрабатываемый язык пользователя и поддерживаются базовыми программными средствами PLAN'a. Совокупность этих формализмов будем называть *абстрактом языка пользователя*.

Как определено в абстракте языка пользователя, задание пакету является последовательностью *операторов*, внешнее представление каждого из которых имеет вид:

<команда> [. <данные>];

Команда может содержать от 1 до 9 фраз, которые именуют отдельные макрооперации процесса обработки данных, реализуемые пакетом, и определяются с помощью языка генерации. Абстракт языка пользователя не накладывает никаких ограничений на состав и порядок следования фраз в команде. Это дает возможность разработчику пакета самому указать правила составления пользовательских команд, обеспечивающие семантическую корректность и наглядность операторов задания.

Фраза, в свою очередь, является фиксированной последовательностью от 1 до 5 слов, разделенных пробелами. И, наконец, *слово* – это непустая последовательность произвольного числа букв, из которых значащими являются только три первых.

Совокупность данных, рассматриваемых в предметной области пакета, при использовании PLAN'a понимается как множество величин, идентифицируемых по именам. Имя величины – это слово, которое должно быть специфицировано в определении одной из фраз языка пользователя.

В *разделе данных* оператора языка пользователя можно задать присваивание начальных значений некоторым величинам, используемым при выполнении этого оператора. Причем начальное значение величины может быть задано как в виде

константы (числовой, логической или литерной), так и в виде арифметического или логического *выражения*, в котором в качестве операндов используются другие величины. Для этого в абстракт языка пользователя включено специальное языковое подмножество, позволяющее описывать простые процессы вычислений, результаты которых могут быть присвоены тем или иным величинам.

Поясним введенные понятия следующими примерами.

**ЧАСТОТНАЯ ДИСПЕРСИЯ, ОПЫТ 10, БАЗА=ГЕНЕРАТОР/2;
АМПЛИТУДНАЯ ДИСПЕРСИЯ, ОПЫТ 15, ШУМ=ФОН-(ВОЗМУЩ*2);**

В приведенных операторах использованы команды, состоящие из двух фраз, вторая из которых ДИСПЕРСИЯ. Операторы выполняются над данными, полученными при проведении соответственно 10-го и 15-го эксперимента. Начальные значения величин БАЗА и ШУМ заданы в виде выражений, которые будут вычислены непосредственно перед выполнением оператора.

Абстракт языка пользователя является языковой составляющей базового системного обеспечения, т.е. той части PLAN'a, которая непосредственно переходит в системное обеспечение каждого создаваемого пакета. Что же касается программных компонентов базового обеспечения, то основными среди них являются *монитор, интерпретатор языка пользователя и библиотека сервисных модулей*. Эти компоненты обеспечивают единообразный стиль ведения работ при создании, эксплуатации и модификации пакета. Они определяют (своими внешними спецификациями) регламент внутренней модуляризации и форму хранения функционального наполнения, способ организации межмодульного интерфейса, возможные режимы выполнения заданий. Так, базовые компоненты PLAN'a предполагают, что:

- языками функционального наполнения являются Фортран и автокод БЕМШ;
- функциональные модули хранятся в виде объектных модулей;
- межмодульный интерфейс по управлению осуществляется по схеме косвенной коммутации (см.п.1.7) с помощью сервисных модулей;
- межмодульный интерфейс по данным является жестким и реализуется через параметры операторов CALL и глобальные переменные, описываемые в операторах COMMON;
- операторы задания могут выполняться как в закрытом, так и в интерактивном режиме, причем возможно чередова-

ние этих режимов в течение сеанса обработки одного задания.

Базовые программные средства PLAN'a обеспечивают также и поддержку межоператорного интерфейса по данным, который реализуется через так называемый *массив связи*, размещаемый в непомеченном COMMON-блоке. Регламент модуляризации функционального наполнения жестко фиксирует внутреннюю структуру общей области памяти, отводимой для этого блока, но вместе с тем и предоставляет разработчикам пакета полную свободу при размещении данных внутри массива связи. Размещая в массиве связи все величины, на которые пользователь может ссылаться в операторах языка заданий, разработчики пакета обеспечивают общность доступа к этим величинам как на уровне языка заданий, так и на уровне базового языка программирования.

Кроме того, в PLAN'e предусмотрена возможность установить иерархию команд с точки зрения организации доступа к величинам, размещаемым в массиве связи. При использовании иерархически упорядоченных команд очередному оператору доступно только то содержимое массива связи, которое было получено непосредственно после выполнения ближайшего предшествующего оператора с командой более высокого уровня. Введение иерархии повышает гибкость и надежность интерфейса между операторами, а также позволяет локализовать последствия ошибки, обнаруженной при выполнении некоторого оператора.

Отметим основные функции базовых программных компонентов PLAN'a.

Монитор осуществляет связь пакета с штатным программным обеспечением, организует выполнение расчетной цепочки, соответствующей очередному оператору задания, обнаруживает ошибочные ситуации в ходе выполнения цепочки и выдает диагностические сообщения. При условии полного соблюдения регламента модуляризации разработчиками функционального наполнения монитор обеспечивает также ликвидацию последствий обнаруженных ошибок.

Интерпретатор языка пользователя осуществляет ввод операторов задания с устройства ввода, производит в соответствии с описанием раздела данных оператора присваивание начальных значений величинам и, используя словарь фраз, который создается с помощью инструментальных средств PLAN'a, формирует расчетную цепочку, реализующую команду оператора.

Библиотека сервисных модулей предназначена для обеспечения взаимодействия функциональных модулей, создания

и обслуживания файлов, используемых функциональными модулями, и организации проблемно-ориентированной подсистемы обнаружения и диагностики ошибок, возникающих в ходе выполнения функциональных модулей.

5.2.2. Инструментальные средства. Развитие абстракта языка пользователя в язык заданий конкретного пакета осуществляется с помощью инструментальных средств PLAN'a: *языка генерации* и *процессора языка генерации*. Эти средства автоматизируют процесс создания языка заданий и дают возможность разработчику пакета отразить в определяемом языке специфику конкретной прикладной деятельности.

Язык генерации имеет декларативный характер и никак не связан с предметной ориентацией создаваемых пакетов. На этом языке записываются определения всех фраз, которые будут использоваться при формировании операторов заданий. Основными объектами языка генерации являются фразы, имена функциональных модулей и величин.

По существу, процесс определения языка заданий пакета заключается в формировании *словаря фраз*, который составляется с помощью процессора языка генерации. Для того, чтобы этот процессор имел возможность оперировать определениями фраз, в абстракте языка пользователя определены три инструментальные фразы: ADD PHRASE, DELETE PHRASE и ALTER PHRASE.

Используя эти фразы, можно включать в задание следующие инструментальные операторы:

ADD PHRASE: <определение фразы> ;

(Этот оператор обеспечивает добавление новой фразы в словарь. Если фраза уже определена, то добавление не производится.)

DELETE: <фраза> ;

(Этот оператор удаляет определение указанной в нем фразы из словаря.)

ALTER PHRASE: <определение фразы> ;

(Этот оператор заменяет старое определение фразы на новое. Если фразы в словаре не было, то действие этого оператора эквивалентно выполнению оператора с командой ADD PHRASE.)

Заметим, что в инструментальных операторах команда отделяется от данных двоеточием, которое для интерпретатора языка пользователя служит признаком того, что раздел данных текущего оператора обрабатываться не должен.

Определение фразы содержит фразу, описание семантики фразы и описания связанных с этой фразой величин, на которые можно ссылаться как на ключевые параметры в разделах данных тех операторов, где используется определяемая фраза.

Общий формат определения фразы:

<фраза> , <описания> ;

Внешний синтаксис раздела <описания> можно определить следующей конструкцией:

<раздел описания семантики> [, <описание уровня>]

$$\left[, \left\{ \begin{array}{l} \text{<раздел описания семантики>} \\ \text{<описание величины>} \end{array} \right\} \dots \right]$$

(В этой нотации большие фигурные скобки, охватывающие несколько строк, означают произвольный выбор одной из этих строк. Следующее за скобками многоточие означает повторение их содержимого произвольное число раз.)

Семантика фразы описывается путем задания расчетной цепочки, которая должна быть выполнена при ссылке на данную фразу. *Раздел описания семантики* определяет некоторый фрагмент этой цепочки и имеет формат:

$$\left\{ \begin{array}{c} \text{PROGRAMS} \\ \text{<логическое выражение>} \end{array} \right\} \text{ 'список модулей'}$$

Список модулей представляет собой непустую последовательность разделенных запятыми имен функциональных и сервисных модулей. В случае, когда раздел описания семантики начинается со служебного слова PROGRAMS, указанный в нем список модулей всегда включается в расчетную цепочку. При задании в определении фразы нескольких возможных вариантов расчетной цепочки используется второй формат раздела описания семантики. В этом случае указанный в разделе описания семантики список модулей включается в расчетную цепочку только тогда, когда результат вычисления логического выражения равен ИСТИНА.

Описание уровня – это синтаксическая конструкция вида

LEVEL <номер уровня>

которая используется для указания уровня (в диапазоне от 1 до 4) фразы в иерархии множества фраз, определяемых при разработке пакета. Уровень фразы существует только для определения уровня команды, который равен уровню самой первой входящей в нее фразы. При опущенном описании уровня фразе присваивается так называемый пустой уровень.

Описание величины, во-первых, определяет идентификатор, посредством которого можно ссылаться на эту величину как в определении фразы, так и в операторах задания, содержащих определяемую фразу, и, во-вторых, задает ряд атрибутов величины, обеспечивающих возможность организации межоператорного интерфейса по данным и согласование представления значений этой величины в задании пакету и в функциональных модулях.

Общая форма описания величины:

[<тип>] [<масштаб>] [<адрес>]
<имя> [<значение по умолчанию>]

Тип определяет значения, которые может принимать описываемая величина, и форму их внутреннего представления. Если тип явно не определен, то по умолчанию величине присваивается тип REAL (в смысле Фортрана).

Масштабирование применяется для того, чтобы дать возможность пользователю оперировать с величинами в удобных для него единицах измерения, отличных от тех, на которые рассчитаны функциональные модули.

Все величины, которые могут использоваться в операторах языка заданий, размещаются в массиве связи. *Адрес* указывает номер того элемента массива связи, в котором будут помещены значения величины (или первой ее компоненты, если величина является одномерным массивом). Переменная функционального модуля, соответствующая описываемой величине, должна быть приписана с помощью средств базового языка программирования к той же позиции непомеченного COMMON-блока, к которой приписан элемент массива связи с указанным в описании величины номером. Адрес может быть задан либо в виде целой константы, либо в виде арифметического выражения. Если адрес не задан, то считается, что величина приписана к элементу, который

следует непосредственно за элементом, указанным в предыдущем описании величины.

Имя величины – это слово, которое служит для идентификации величины в языке заданий и языке генерации.

Значение по умолчанию – это стандартное значение величины, заносимое в соответствующий элемент массива связи, прежде чем в него будет занесено начальное значение величины, т.е. значение, которое может быть присвоено величине в разделе данных оператора, использующего определяемую фразу. Значение по умолчанию может быть задано либо в форме константы, либо в форме выражения.

Операторы, задающие определения трех фраз (ДИСПЕРСИЯ, ЧАСТОТНАЯ и АМПЛИТУДНАЯ), которые были использованы в командах операторов, п. 5.2.1, могут быть записаны следующим образом:

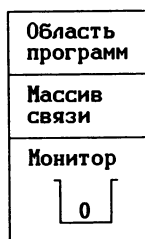
```
ADD PHRASE: ДИСПЕРСИЯ, I(1) ОПЫТ, PROGRAMS 'MOD001';  
ADD PHRASE: ЧАСТОТНАЯ, P+2 (2) БАЗА 30, ГЕНЕРАТОР,  
PROGRAMS 'MOD010,MOD011';  
ADD PHRASE: АМПЛИТУДНАЯ, (2) ШУМ=ШУМ*2, ФОН 10,  
ВОЗМУЩ, PRO 'MOD010,MOD030', (ВОЗМУЩ>ФОН*0.5) 'MOD020';
```

Описание величины ОПЫТ включает указание типа (I – целый) и адреса (1). Для величины БАЗА в определении фразы ЧАСТОТНАЯ указано масштабирование значения в 100 раз (P+2) и значение по умолчанию (30). Вычисление значения величины ШУМ задано в определении фразы АМПЛИТУДНАЯ арифметическим выражением. Заметим, что адреса для величин ГЕНЕРАТОР, ФОН и ВОЗМУЩ не указаны и определяются по умолчанию. Семантика команды ЧАСТОТНАЯ ДИСПЕРСИЯ состоит в том, что программа MOD010 осуществляет вывод данных, полученных в опыте с номером ОПЫТ, MOD011 вычисляет по ним среднюю частоту (см. определение ЧАСТОТНАЯ), а затем MOD001 рассчитывает ее дисперсию. У команды АМПЛИТУДНАЯ ДИСПЕРСИЯ семантика частично совпадает с семантикой предыдущей команды (выполнение MOD001 и MOD010), но, во-первых, при слишком большом значении ВОЗМУЩ, которое проверяется в логическом выражении, задается сглаживание (модуль MOD020), и, во-вторых, вместо частоты вычисляется амплитуда (MOD030).

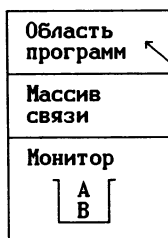
5.2.3. Общая схема функционирования. Рассмотрим основные этапы функционирования пакета, созданного с помощью PLAN'a (точнее говоря, с помощью его версии PLAN-

Инициализация

(A)



(B1)



Стек не пуст

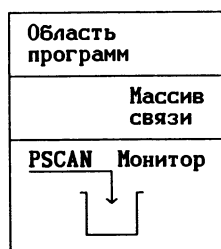
(B2)



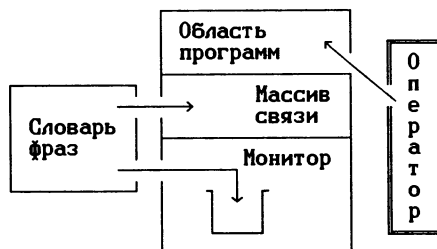
Библиотека

Стек пуст

(C1)

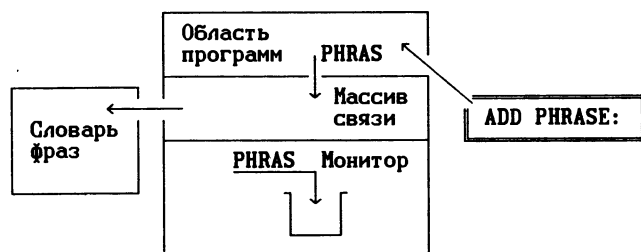


(C2)



Команды модификации словаря

(D)



PSCAN – интерпретатор языка пользователя,

PHRAS – процессор языка генерации

Рис.5.2. Функционирование PLAN-БЭСМ-6

БЭСМ-6), и работу основных программных компонентов анализатора на этих этапах (рис.5.2).

Запуск пакета осуществляется через мониторную систему Дубна, функционирующую в рамках операционной системы Диспак. Поэтому набор предложений запуска пакета состоит из паспорта ОС Диспак и последовательности управляющих предложений системы Дубна, в которую входит специальное управляющее предложение, обеспечивающее вызов пакета. Следом за предложением вызова пакета должны располагаться тексты операторов задания, которые последовательно вводятся и исполняются. Среди этих операторов могут быть и системные операторы, переключающие пакет на режим ввода с терминала. После такого переключения операторы задания можно последовательно вводить с терминала. Предусмотрены также системные операторы, осуществляющие обратное переключение.

По умолчанию вывод результатов счета и системных диагностических сообщений производится на устройство печати. Специальные операторы позволяют переключить вывод данных и диагностики на терминал.

Используемые при работе пакета модули базового системного обеспечения и модули функционального наполнения должны находиться в библиотеках, с которыми работает динамический загрузчик системы Дубна. Словарь фраз пакета располагается на магнитной ленте (диске), которая должна быть заказана в паспорте ОС Диспак.

Перейдем к рассмотрению отдельных этапов, изображенных на рис.5.2.

(А) Программа инициализации пакета, вызванная системой Дубна, загружает монитор и выделяет начальный участок памяти для массива связи. Вся оставшаяся часть памяти (на рисунке она обозначена как область программ) в последующем используется для размещения функциональных и системных модулей, реализующих операторы задания.

(В) Монитор является единственным постоянно находящимся в оперативной памяти программным компонентом системного наполнения пакета и занимает около 2К слов. Одной из основных функций монитора является организация выполнения расчетной цепочки, реализующей оператор задания. При этом он использует *стек управления*, в котором размещаются имена модулей, входящих в расчетную цепочку.

В начале обработки очередного оператора интерпретатором языка пользователя в стек управления заносятся расчетные цепочки, выбираемые из определений фраз, входящих в данный оператор. Кроме того, имена модулей могут

заноситься в стек и в процессе исполнения расчетной цепочки, если межмодульный интерфейс по управлению осуществляется через сервисные модули PLAN'a.

Очередным подлежащим выполнению модулем всегда является модуль, имя которого указано в вершине стека управления. Загрузка модулей в память и редактирование внешних связей осуществляется стандартным загрузчиком системы Дубна. После загрузки модуля его имя удаляется из стека и иницируется его выполнение. В соответствии с регламентом модуляризации каждый модуль завершает свою работу передачей управления монитору, и это обеспечивает выполнение всех модулей расчетной цепочки.

(С) При пустом стеке управления (что имеет место сразу же после загрузки монитора или после завершения обработки очередного оператора) монитор передает управление интерпретатору языка пользователя. Интерпретатор вводит очередной оператор задания с текущего устройства ввода, производит его проверку и интерпретацию. Для этого он находит в словаре фраз пакета определения всех фраз, использованных в команде оператора. Из определений фраз извлекаются содержащиеся в них задания расчетных цепочек и помещаются в стек управления. По окончании интерпретации оператора управление возвращается монитору.

(D) Процессор языка генерации работает при определении и модификации языка заданий пакета. В его функции входит проверка правильности определения фразы, создание его внутреннего представления и запись этого представления в словарь фраз пакета. Работа процессора языка генерации иницируется в результате интерпретации определений системных фраз ADD PHRASE, DELETE PHRASE и ALTER PHRASE, расчетные цепочки которых содержат ссылки на процессор языка генерации.

5.3. Разработка пакетов на основе анализатора PLAN-БЭСМ-6

Хотя анализатор PLAN-БЭСМ-6 и является проблемно-независимой инструментально-базовой системой, тем не менее вряд ли имеет смысл говорить о единой процедуре разработки пакетов на его основе. Содержание и регламент такой процедуры в значительной степени зависят и от прикладной деятельности, которая будет обслуживаться создаваемыми пакетами, и от чисто субъективных факторов, привносимых разработчиками и пользователями пакетов. Поэтому мы

рассмотрим здесь только основные принципы разработки пакетов, являющихся многоцелевыми системами обработки экспериментальных данных. Эти принципы сжато отражают технологию использования PLAN-БЭСМ-6 аналитиками и прикладными программистами ИПМ АН СССР, имеющими уже значительный опыт разработки многоцелевых систем с помощью штатных средств программного обеспечения машины БЭСМ-6 [124].

Разработка пакета на основе анализатора PLAN-БЭСМ-6 проводится по следующей схеме (см. рис.5.3).

(А) В соответствии с постановкой задачи автоматизации обработки данных физических экспериментов конкретного типа составляется неформальная модель процесса сплошной математической обработки экспериментальных данных, которая отражает логическую структуру, основные этапы (подзадачи) и объекты этого процесса. Далее каждый из этапов детализируется с целью выявления множества базовых макроопераций, соответствующих минимальным единицам действия, которые необходимо иметь возможность задавать на уровне языка пользователя пакета.

(В) Создается терминологический словарь пакета: выделенным объектам и макрооперациям процесса обработки экспериментальных данных ставятся в соответствие величины и фразы языка пользователя, которые идентифицируются имеющими содержательный смысл словами и терминами предметной области пакета.

(С) Используя логическую схему процесса обработки, на множестве введенных фраз и величин задают правила составления пользовательских команд, каждая из которых обычно соответствует одной из подзадач обработки экспериментальных данных.

(D) Устанавливаются связи между величинами и командами (фразами) языка заданий пакета. Эти связи должны показывать, какие величины используются при выполнении данной команды и как ее выполнение зависит от результатов выполнения других команд. В дальнейшем схемы взаимодействия команд (межоператорный интерфейс по данным) формализуются (на шаге (G)) с помощью аппарата уровней.

(Е) Проектируются алгоритмы макроопераций и выбирают представления объектов реализуемого пакетом процесса обработки данных. Определяются такие характеристики объектов, как формат машинного представления, диапазоны изменения, стандартные значения. На этом же этапе выявляются величины, которые должны быть доступны пользова-



Рис 5.3. Разработка пакетов на основе PLAN-БЭСМ-6

телю, и производится привязка этих величин к элементам массива связи.

(F) Программируются функциональные модули пакета. На этом этапе могут использоваться различные сервисные модули, входящие в базовое системное обеспечение. Отладка и тестирование функциональных модулей заканчивается созданием библиотеки загрузочных модулей функционального наполнения пакета.

(G) С помощью языка генерации на основе сведений, полученных на предыдущих этапах, определяются фразы языка пользователя. Определения фраз помещаются в словарь фраз пакета с помощью инструментальной команды ADD PHRASE. Собственно говоря, процесс генерации языка заданий и выражается в выполнении соответствующей последовательности операторов вида:

ADD PHRASE: <фраза> , <описания> ;

При этом на этапе генерации языка заданий взаимодействие разработчика пакета с анализатором PLAN-БЭСМ-6 протекает точно так же, как и взаимодействие пользователя с пакетом при решении задач.

В соответствии с рассмотренными принципами в ИПМ АН СССР с помощью анализатора PLAN-БЭСМ-6 был разработан пакет КОРПУСКУЛА [124], ориентированный на решение задач определения пространственного распределения ионной температуры лазерной плазмы на основе времяпролетных измерений потока продуктов ядерных реакций. Трудоемкость разработки этого пакета (без учета затрат на программирование функционального наполнения) – 0.5 человекогода.

5.4. Внешнее информационное обслуживание

Как уже отмечалось (см. п. 5.1), средства внешнего информационного обслуживания пакетов (систем) сплошной математической обработки экспериментальных данных служат преимущественно для выдачи по запросам пользователей сведений о назначении и функциональных возможностях пакета, о наборе команд (фраз) языка заданий, их синтаксисе и семантике. Причем, поскольку такие пакеты ориентированы на эксплуатацию их конечными пользователями различной квалификации, внешнее информационное обслуживание должно обеспечивать возможность обучения в режиме диалога пользователя – новичка и в то же время не обременять излишней опекой и «говорливостью» пользователя, уже имеющего навыки работы с пакетом, но не знающего

(или забывшего) некоторые конкретные детали языка заданий или регламента работы с пакетом.

Для обеспечения такого информационного сервиса в языке заданий пакетов программ, разрабатываемых на основе анализатора PLAN-БЭСМ-6, кроме *команд обработки*, т.е. задающих основные этапы процесса обработки экспериментальных данных, включаются так называемые *информационные команды*. Набор этих команд определяется на основе некоторого древовидно-структурированного сценария обучения, предлагаемого пользователям пакета его разработчиками.

Каждой позиции дерева обучения соответствует определенная информационная команда. Использование информационных команд возможно как в режиме обучения (т.е. в рамках прохождения по дереву обучения от его корня), так и в режиме частных справок по конкретным запросам пользователя вне зависимости от места соответствующих команд в дереве обучения. Оба режима информационного обслуживания допускаются в любой момент работы с пакетом, когда разрешен ввод очередного оператора задания. Отметим, что выполнение информационных команд обеспечивает только выдачу интересующих пользователя сведений и никак не влияет на ход вычислений, проводимых в соответствии с операторами, собственно задающими процесс обработки экспериментальных данных.

В качестве примера такого подхода рассмотрим организацию внешнего информационного обслуживания пользователей пакета КОРПУСКУЛА, для которого принят сценарий обучения, показанный на рис.5.4 [129].

В начале работы с пакетом выдается следующее сообщение:

**ПАКЕТ КОРПУСКУЛА К РАБОТЕ ГОТОВ
ЕСЛИ ВЫ НЕЗНАКОМЫ С ПАКЕТОМ ВЫДАЙТЕ ОПЕРАТОР НАУЧИ;**

После этого пользователь, желающий использовать режим обучения, вводит оператор НАУЧИ. В ответ на такой запрос пользователю сообщается краткая общая справка о пакете, а также указываются операторы, которыми он может воспользоваться для того, чтобы перейти к знакомству с пакетом по следующим трем информационным рубрикам: назначение пакета, правила задания операторов, набор команд языка заданий пакета.

Следуя далее по поддереву каждой из этих рубрик, пользователь имеет возможность получить все сведения, необходимые для составления задания, реализующего реше-



Рис 5.4. Сценарий обучения работе с пакетом КОРПУСКУЛА

ние стоящей перед ним задачи (разумеется, если она относится к классу задач, обслуживаемых пакетом). При последующих сеансах работы с пакетом пользователь может и не придерживаться сценария обучения, а выдавать информационные запросы для получения частных справок, связанных с контекстом конкретного сеанса.

Такой подход к организации внешнего информационного обслуживания наряду с тем, что он допускает возможность сколь угодно глубокой детализации сведений о пакете, имеет еще два важных достоинства.

Во-первых, он прост в реализации, которая сводится к разработке определений соответствующих информационных фраз и написанию достаточно простых функциональных модулей, осуществляющих выдачу текстов сообщений для пользователя. Причем то обстоятельство, что PLAN позволяет применять многофразные команды, дает возможность сформировать набор необходимых информационных команд на базе сравнительно небольшого числа информационных фраз. Так, определив соответствующим образом используемые в качестве префиксов фразы типа СИНТАКСИС, СЕМАНТИКА, ЗНАЧЕНИЕ ПО УМОЛЧАНИЮ и т.п. и сочетая их с командами обработки, можно ввести в язык заданий пакета множество легко мнемонически воспринимаемых информационных команд, правило записи которых имеет вид:

<префикс> <команда обработки>

Например:

СЕМАНТИКА ЧАСТОТНАЯ ДИСПЕРСИЯ

или

ЗНАЧЕНИЯ ПО УМОЛЧАНИЮ АМПЛИТУДНАЯ ДИСПЕРСИЯ

Во-вторых, рассматриваемый подход с системной точки зрения инвариантен относительно тематики информационного обслуживания. Действительно, легко видеть, что специфика тематики отражается только в языке заданий и функциональном наполнении пакета и никак не сказывается на системном наполнении. Это обстоятельство дает возможность применять такой подход для организации информационного обслуживания с различной тематической ориентацией (например, для выдачи учебных сведений или частных справок не только о языке заданий, но и о функциональном наполнении) и для различных пакетов.

Глава 6

ПАКЕТЫ ДЛЯ КОНКРЕТНЫХ ПРИЛОЖЕНИЙ

6.1. Пакет КРИТ: многокритериальная оптимизация

Пакет КРИТ [130] предназначен для решения многокритериальных задач оптимизации объектов машиностроения. Пакет предоставляет набор средств, поддерживающих определенную методику оптимизации и избавляющих пользователя пакета от рутинной работы по реализации многочисленных сервисных программ. Конструктор – пользователь пакета – программирует (в рамках определенных интерфейсных соглашений) модель исследуемого объекта, после чего с помощью программ пакета КРИТ выполняется оптимизация этого объекта по ряду выбранных конструктором критериев.

6.1.1. Многокритериальная оптимизация машин. Задачи оптимизационного проектирования машин всегда многокритериальны: желательно уменьшить себестоимость, увеличить надежность, уменьшить материалоемкость, увеличить КПД и т.д. Улучшение качества проектируемой машины по одному из таких первичных, понятных конструктору критериев, как правило, сопровождается весьма нетривиальным изменением (обычно ухудшением) показателей по другим критериям. Среда конструирования настолько сложна, что дать априори какое-либо формализованное интегральное определение оптимальности машины конструктор просто не в состоянии.

Однако подавляющее большинство существующих математических методов оптимизации предназначены для отыскания оптимального значения одной функции – одного критерия. Поэтому чаще всего пытаются неоправданными упрощениями свести многокритериальную задачу к однокритериальной. По убеждению авторов работы [131], большинство неудачных решений берет свое начало именно на этом этапе, так как если сформулированная математическая задача не адекватна исходной задаче, то никакой метод оптимизации положения не спасет: найденное «оптимальное решение» будет плохим.

Кто же должен давать математическую постановку задачи? Обычно считают, что это дело конструкторов, которые хорошо разбираются в существе дела. Однако в действительности, когда речь идет о задачах со многими параметрами и с несколькими критериями, поставить такого рода математическую задачу очень трудно, ибо конструктор обычно имеет достаточно хорошее представление о допустимых пределах изменения каждого параметра, но не знает возможностей всех критериев, и необходим предварительный расчет для выяснения этих возможностей.

Сформулируем мнение авторов работы [131] по поводу проблемы выбора оптимальных параметров.

Во-первых, необходимо рассматривать задачи оптимального проектирования как многокритериальные, а не сводить их к однокритериальным, так как это, как правило, приводит к серьезному огрублению задачи, искажению ее существа, а следовательно, к неоправданной замене одной задачи другой. Что касается конструкторов, то они всегда были рады поддержать такую точку зрения.

Во-вторых, не следует стремиться к полной автоматизации процесса выбора оптимальных параметров: выбор должен осуществлять сам конструктор, но с помощью ЭВМ. Поэтому наиболее рациональными являются алгоритмы, поддерживающие диалог человека с вычислительной машиной.

Проектирование реальных объектов с учетом многих критериев качества обычно имеет характер эвристического итерационного процесса: конструктор, рассматривая различные варианты модели, оценивает результаты, уточняет постановку задачи, затем снова решает ее и анализирует новые варианты. В процессе проектирования нередко меняются взгляды на значимость отдельных критериев. И это продолжается до тех пор, пока конструктор не сочтет, что пришло время остановиться: найдено то, что ему нужно.

Пакет КРИТ автоматизирует этот процесс, поддерживая диалог конструктора с ЭВМ: при помощи ЭВМ формируется множество допустимых решений, среди которых конструктор выбирает наилучшее. Формирование множества допустимых решений базируется на систематическом просмотре многомерных областей. В качестве пробных точек пространства параметров берутся точки равномерно распределенных последовательностей.

Особенность метода состоит в том, что диалог конструктора с ЭВМ протекает в очень благоприятных для конструктора условиях: он оперирует привычными для него величинами — значениями критериев — и не должен «комбиниро-

вать», т.е. гадать, какой выигрыш по одним критериям могут дать уступки по другим критериям; это выясняется в процессе диалога.

6.1.2. Организация исследования. Введем некоторые понятия, используемые в дальнейшем изложении. Параметрами α_i ($1 \leq i \leq N$) будем называть те величины, которые конструктор может варьировать для получения оптимального решения. Пространством параметров будем называть N -мерное пространство, состоящее из точек A с координатами $(\alpha_1, \dots, \alpha_N)$.

Как правило, конструктор может указать разумные пределы изменения каждого из параметров, которые мы будем называть параметрическими ограничениями:

$$\alpha_i^* \leq \alpha_i \leq \alpha_i^{**} \quad (1 \leq i \leq N) \quad (1)$$

В результате получается N -мерный прямоугольный параллелепипед в пространстве параметров. Объем этого параллелепипеда обозначим через V . Программа, моделирующая исследуемый объект, должна по заданной совокупности значений параметров модели рассчитать значения ряда функционалов φ_j ($1 \leq j \leq M$). Каждый из функционалов имеет очевидный для конструктора физический смысл (мощность, стоимость, вес и т.д.). Конструктор должен указать, какие функционалы будут выступать в качестве критериев оптимизации. Остальные вычисляемые функционалы станут рассматриваться как псевдокритерии — их значения интересуют конструктора, но в процессе определения оптимальных значений параметров они не участвуют. Написав моделирующую программу и определив критерии оптимизации, можно переходить к вычислению допустимых точек в пространстве параметров, которое включает в себя этапы расчета, выбора критериальных ограничений и обработки.

На этапе *расчета* программы пакета КРИТ выбирают K пробных точек A_1, \dots, A_K , равномерно распределенных в области, заданной ограничениями (1). В каждой из пробных точек посредством обращения к моделирующей программе вычисляются значения функционалов φ_j ($1 \leq j \leq M$). Для каждого функционала составляется так называемая таблица испытаний, в которой значения $\varphi_j(A_1), \dots, \varphi_j(A_K)$ расположены в порядке возрастания (убывания) и указаны номера соответствующих пробных точек. Таблица испытаний позволяет увидеть максимум и минимум функционала,

получить представление о частоте тех или иных значений $\varphi_j(A)$.

На этапе *выбора* конструктор, просматривая таблицы испытаний, должен для каждого критерия из числа функционалов φ_j назначить критериальные ограничения (односторонние или двухсторонние), т.е. указать диапазон значений критерия, который он считает приемлемым. При первоначальных расчетах рекомендуется задавать относительно слабые ограничения, с тем чтобы в таблицах испытаний осталось больше допустимых точек.

На этапе *обработки* программы пакета КРИТ отбирают те пробные точки, для которых значения всех критериев попадают в диапазон, указанный конструктором. Если множество отобранных таким образом допустимых точек пусто, то следует вернуть конструктора на этап выбора критериальных ограничений и потребовать от него уступок, т.е. расширения границ диапазонов. Если конструктор продолжает настаивать на своих ограничениях, можно попытаться повторить этап расчета, увеличив число исследуемых пробных точек.

Если и это не помогло, т.е. при большом числе испытаний допустимые точки не обнаруживаются, то имеются серьезные основания считать, что выбранные критериальные ограничения несовместны. Конечно, нельзя категорически исключить возможность того, что в некоторой точке A' , отличной от A_1, \dots, A_K , получаются удовлетворяющие конструктора значения критериев. Однако, даже если такая точка A' существует, то из-за равномерности распределения пробных точек ее окрестность, в которой сохраняется «замечательное» свойство, очень мала (объем ее порядка V/K) и практически система, соответствующая A' , будет неустойчивой (не конструктивной).

Если же множество допустимых точек не пусто, то можно переходить к более подробному анализу этих точек, на основе которого и решается исходная задача оптимизации машины.

Здесь может сложиться впечатление, что при проведении расчетов по рассмотренной методике средства пакета КРИТ играют весьма скромную роль. На самом деле это далеко не так.

Прежде всего отметим, что узловым моментом в данной методике является равномерность распределения пробных точек в N -мерном пространстве параметров. Выяснилось, что большинство существующих датчиков случайных чисел не удовлетворяют в полной мере требованию равномерности и

необходим специальный алгоритм генерации пробных точек. Программа, реализующая этот алгоритм, разумеется, включена в пакет КРИТ, как и многие другие общие для оптимизационных расчетов программы. Тем самым не только экономятся усилия по программированию конкретных оптимизационных задач, но и, что не менее важно, у пользователя пакета появляется вполне обоснованная уверенность в том, что он ничего не упустил и правильно интерпретирует предписания данной методики.

Кроме того, используемый пакетом общий каркас оптимизационной программы содержит в себе удобные гнезда для конкретных фрагментов программ моделирования. Располагая вновь создаваемые программы в этих гнездах, пользователь получает в свое распоряжение богатый арсенал алгоритмических находок, собранных разработчиками пакета из различных решавшихся ранее оптимизационных задач.

6.1.3. Каркас оптимизационной программы. Предписываемый пакетом КРИТ каркас оптимизационной программы изображен на рис.6.1. В каркасе можно выделить две части: ядро пакета, недоступное пользователю для модификации, и оболочку, в гнездах которой располагаются программы пользователя, реализующие конкретную оптимизируемую модель. Рассмотрим назначение отдельных гнезд каркаса.

Расчет начинается с главной программы (MAIN). Здесь задаются параметры расчета, значения границ изменения параметров модели, ограничения на критерии, распределяется память для рабочих массивов и др.

Программа ядра – COTROL – осуществляет организацию основного цикла работы пакета.

Вначале выполняется программа оболочки INITIAL, инициализирующая расчет. Инициализация может включать в себя ввод начальных данных, вычисление констант, очистку массивов, открытие файлов и др. Предполагается, что программа INITIAL работает один раз на расчет.

Дальнейшая работа пакета производится в рамках цикла, число повторений которого определяется числом требуемых испытаний. При каждом выполнении тела цикла производится вычисление значений координат очередной пробной точки по значениям стандартной равномерно распределенной в N -мерном единичном кубе квазислучайной последовательности. В простейшем случае вычисление сводится к отображению множества точек единичного N -мерного куба в точки N -мерного параллелепипеда, границы которого были заданы

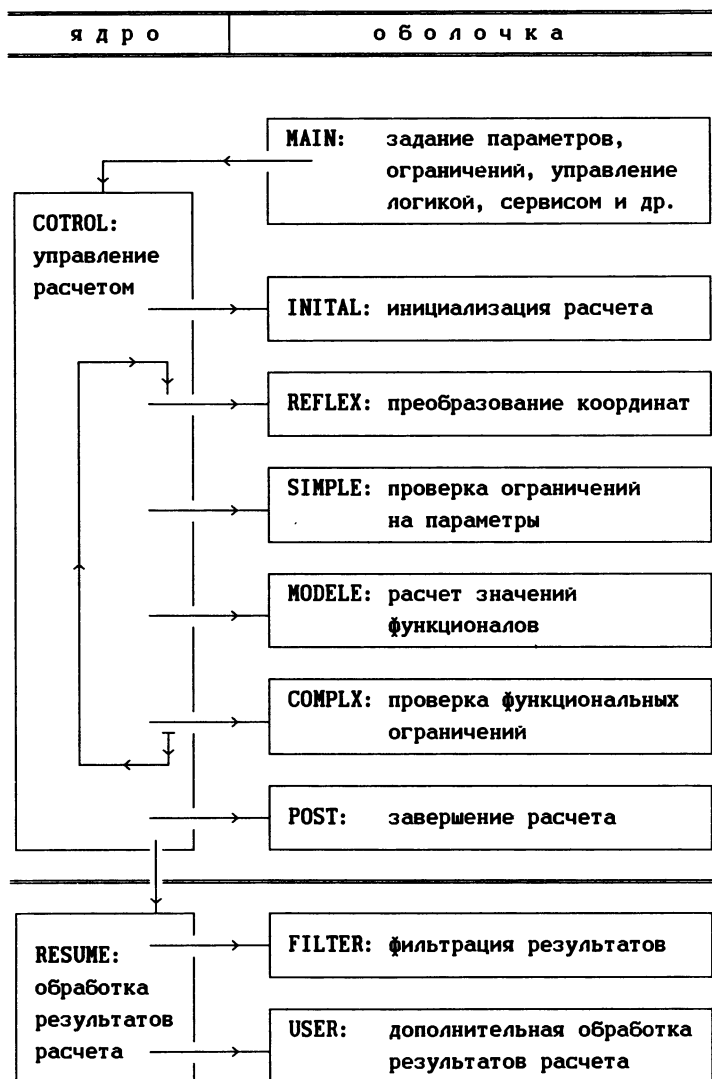


Рис 61 Каркас оптимизационной программы

в виде параметрических ограничений. Вычисление значений координат пробных точек производится в программе, размещаемой в гнезде оболочки REFLEX.

После того, как очередная пробная точка получена, нужно проверить, удовлетворяет ли она простым функциональным ограничениям на параметры. Дело в том, что в конкретных приложениях форма области определения задачи как правило отличается от N -мерного параллелепипеда – в области могут быть «вырезаны» L -мерные полосы, выколоты точки и т.д. В этом случае форма области может быть задана в виде алгебраических неравенств и соотношений, называемых в дальнейшем простыми функциональными ограничениями. Для получения координат пробной N -мерной точки сначала получают значения координат в объемлющем область определения N -мерном параллелепипеде, а затем проверяют функциональные ограничения. Если точка попала в область определения, то она пригодна для дальнейших расчетов. Программа оболочки, осуществляющая проверку пригодности пробной точки, размещается в гнезде SIMPLE.

Если точка признается непригодной, то пакет переходит к выбору следующей пробной точки. Если точка признана пригодной (т.е. она принадлежит области определения задачи), то выполняется программа расчета значений функционалов в полученной пробной точке.

Программа расчета – MODELE – должна быть организована так, чтобы она допускала многократное обращение. Если вычисление значений в одной точке занимает много времени, то, чтобы сократить возможный ущерб от повторного счета при сбоях, рекомендуется использовать возможности пакета по созданию контрольных точек через любое заданное число шагов.

Полученные значения функционалов могут быть дополнительно исследованы. В некоторых задачах существуют требования, налагаемые на комбинации значений функционалов. Например, два функционала не должны сильно отличаться друг от друга, сумма значений других функционалов не должна превышать значения некоторой константы и др. По существу, введение такого требования эквивалентно введению нового «псевдокритерия», для которого заданы ограничения. Однако конструктору может быть неудобно вводить новый псевдокритерий – в этом случае более естественно иметь средство проверки значений самих функционалов. Проверка осуществляется в программе COMPLX. Если значения функционалов в пробной точке подходят, то эти значения запоминаются пакетом и подлежат дальнейшей

обработке. Заметим, что в отличие от программы проверки простых функциональных ограничений (SIMPLE), программа проверки «сложных» функциональных ограничений на критерии (COMPLX) выполняется после вычисления значений функционалов. И простые, и сложные ограничения могут отсутствовать.

Выполнив рассмотренные выше действия, пакет переходит к вычислению следующей точки.

После того, как проведено требуемое число испытаний, пакет выполняет программу оболочки POST, завершающую расчет модели. Завершающие действия могут включать закрытие файлов пользователя, сбор статистики, печать сообщений и др.

Если конструктор желает совместить расчет с обработкой, то он задает это с помощью определенной комбинации значений управляющих параметров. Обработка результатов испытаний производится программой RESUME и завершается выдачей на печать разнообразных таблиц и графиков. В частности, можно напечатать таблицы испытаний для каждого из функционалов, таблицы значений параметров, таблицы коэффициентов корреляции критериев, значения критериев, удовлетворяющие критериальным ограничениям, сводную таблицу испытаний. Кроме того, можно получить графическое изображение зависимости пар критериев, а также напечатать множество решений, оптимальных по Парето.

Понятие оптимальности по Парето требует некоторых пояснений. Если конструктор желает осуществить выбор параметров, принимая во внимание несколько важнейших критериев φ_{j_l} ($1 \leq l \leq M_1$), то можно облегчить ему этот выбор, исключив из числа пробных точек такие, которые заведомо не могут оказаться наилучшими. Пробная точка A может быть отброшена, если существует пробная точка A' такая, что $\varphi_{j_l}(A') \geq \varphi_{j_l}(A)$ для всех значений l из отрезка $[1, M_1]$ и хотя бы для одного значения l имеет место строгое неравенство. (Предполагается, что критерий φ_{j_l} максимизируется. Если же он минимизируется, то вместо отношения \geq берется отношение \leq .) Паретовскими называются пробные точки, которые остаются после отбраковки всех таких «бесперспективных» точек.

Иногда полезно ограничить обработку определенным подмножеством успешных испытаний. Это может потребоваться для более детального исследования отдельных подобластей пространства параметров, для пересчета значений некото-

рых функционалов и т.д. Отсев ненужной части успешных испытаний выполняет программа оболочки FILTER.

Если конструктору требуется провести дополнительную обработку результатов испытаний, которая не предусмотрена сервисом пакета, то он должен создать соответствующую программу и поместить ее в гнездо USER. Обращение к этой программе производится после выполнения всех принадлежащих ядру пакета программ обработки.

Если конструктор проводит расчет и обработку результатов отдельно, то он должен позаботиться о подключении к задаче устройства внешней памяти, поскольку пакет запоминает результаты испытаний в виде файла на внешней памяти. Номера каналов записи задаются в форме параметров расчета. Имеется возможность разбить расчет на несколько независимых запусков с последующим объединением результатов.

При проведении обработки результатов пакет не использует программы инициализации (INITAL), расчета (MODELE), завершения расчета (POST), проверки ограничений (SIMPLE и COMPLX), но использует программу преобразования координат (REFLEX).

Конкретные интерфейсные соглашения для отдельных гнезд каркаса здесь рассматриваться не будут; они подробно описаны в работе [130]. Отметим только, что все программы ядра пакета КРИТ написаны на языке Фортран-4, и поэтому лучший язык для программ, заполняющих гнезда оболочки, — тоже Фортран-4. Тем не менее программы оболочки могут быть написаны на любом языке, но тогда, вероятно, потребуется создать адаптер, связывающий соответствующие языковые среды.

6.1.4. Особенности реализации. Пакет КРИТ рассчитан на решение многокритериальных оптимизационных задач, удовлетворяющих следующим ограничениям:

- 1) количество параметров не превосходит 30;
- 2) количество вычисляемых функционалов не превосходит 30;
- 3) отношение N -мерного объема множества допустимых решений, расположенного внутри заданного N -мерного параллелепипеда в пространстве параметров, к N -мерному объему этого параллелепипеда не должно быть чрезмерно малым.

Ограничения 1 и 2 связаны с реализацией и могут быть легко сняты. Ограничение 3 может быть проверено в ходе

расчета; оно исключает задачи с очень большим количеством функциональных ограничений.

Как уже упоминалось, программы ядра пакета написаны на языке Фортран-4. Особое внимание уделялось мобильности ядра, благодаря чему в настоящее время пакет широко эксплуатируется на ЭВМ ЕС, БЭСМ-6, СМ-4, VAX, IBM PC и др. Суммарный объем программ ядра пакета – около 5000 операторов Фортрана-4.

Пакет передан в десятки организаций, где с его помощью решаются весьма разнообразные оптимизационные задачи.

6.2. Пакет ОКС

В настоящее время данные о взаимодействии излучения с веществом (сечения взаимодействия), необходимые для расчета полей нейтронов и гамма-квантов в реакторах и радиационной защите, используются в основном в виде групповых макроскопических констант. Эти макроконстанты определяются специальными программами константного обеспечения для вещества каждой зоны реактора или защиты (с учетом состава, температуры и агрегатного состояния) по данным заранее составленной библиотеки микроконстант взаимодействия излучения на каждом энергетическом интервале (группе) с каждым из изотопов.

Получили распространение несколько систем нейтронных и гамма-констант разного происхождения, ориентированных на нужды расчета реакторов различных спектральных классов и радиационной защиты.

Пакет ОКС [132,133] – Объединенная Константная Система – предназначен для объединения наиболее распространенных константных программ на базе проблемно-ориентированного языка. В языке используется устоявшаяся физическая терминология; задание на нем содержит только необходимую для расчета констант информацию и не зависит от конкретной программы, по которой будет вестись расчет констант. Пакет обеспечивает унифицированный интерфейс между программами расчета полей излучения и различными константными программами.

Функциональное наполнение пакета ОКС включает, в частности, константные системы АРМАКО-2F [134], АРМАКО-G [135], базирующиеся на микроданных библиотеки БНАБ-78, 49-групповую систему констант с библиотекой БНД-49 [136], программный комплекс TERMAC [137] для расчета

групповых констант нейтронов в области термализации с энергией 0–5 ЭВ, а также программы ARVES [138] и JAST.

6.2.1. Язык заданий. *Задание* для пакета ОКС представляет собой последовательность операторов. По своему функциональному назначению операторы делятся на *исполняемые* и *неисполняемые*. Исполняемый оператор задает некоторое действие, например, вызов константной программы или преобразование форматов. Неисполняемый оператор – это описание объектов, используемых при работе исполняемого оператора. Порядок выполнения операторов определяется порядком их следования в задании. Следует отметить, что исполняемый оператор может работать только с неисполняемыми операторами, расположенными в задании непосредственно после него до следующего исполняемого оператора; остальные объекты для него недоступны.

Оператор имеет вид

<операция> : <операнды> ;

В поле операции задается служебное слово, являющееся названием операции. Служебное слово – последовательность букв произвольной длины; различаются служебные слова по первым шести символам. В поле операндов задается список параметров операции (они называются также параметрами оператора). Параметры в списке разделяются запятыми. В качестве параметра может употребляться идентификатор или константа. Допускается умолчание параметра. В этом случае соответствующая позиция в поле операндов остается пустой. Запятая или несколько подряд идущих запятых, возникающих в результате умолчания параметров и стоящих перед символом точка с запятой (;), могут быть опущены.

Константы, употребляемые в языке заданий пакета ОКС, записываются в соответствии с синтаксисом Фортрана. *Идентификатор* языка ОКС – это начинающаяся с буквы последовательность (не более чем из шести символов), в которую могут входить буквы, цифры и знак минус (-).

6.2.2. Основные операторы. В данном разделе мы рассматриваем операторы, наиболее часто используемые в заданиях для пакета ОКС. Следует отметить, что по мере эксплуатации пакета набор допустимых операторов постоянно пополняется.

Оператор ZONE – неисполняемый оператор. Он описывает одну физическую зону (вещество), характеризующуюся однородным с физической точки зрения составом. Для того,

чтобы учесть такие случаи, когда изотопы в зоне пространственно разделены и имеют разные температуры, имеется возможность задавать температуру для каждого изотопа отдельно.

Оператор **ZONE** имеет вид

ZONE : <номер зоны>, <общая температура зоны>
{, <имя изотопа>, <ядерная концентрация>,
[<температура изотопа>]} ... ;

(Напомним, что в такой нотации квадратные скобки означают необязательность наличия их содержимого. Фигурные скобки с последующим многоточием означают повторение их содержимого произвольное число раз.) Если температура изотопа не задана, то она по умолчанию считается равной общей температуре зоны. Температура и ядерная концентрация – вещественные числа. Имя изотопа – идентификатор.

Оператор **CONST** вычисляет константы по одной из константных программ, входящих в функциональное наполнение пакета ОКС. Полученному в результате массиву констант присваивается имя, которое задается параметром оператора.

Оператор **CONST** имеет вид

CONST : <имя массива>, <имя программы>, <режим> ;

Имя массива – идентификатор, который будет использоваться при ссылке на полученный массив макроконстант. Имя программы – идентификатор, определяющий версию константной системы, которую следует применить для расчета макроконстант требуемого типа. Режим определяет выбор варианта работы константной программы и задается списком подпараметров, длина которого, а также смысл каждого из элементов зависят от конкретной константной программы. Допустимые значения параметров оператора **CONST** для тех константных систем, которые включены в пакет ОКС, содержатся в инструкции по эксплуатации пакета.

Оператор **DATA** – неисполняемый оператор. Он применяется в тех случаях, когда параметр исполняемого оператора представляет собой массив констант.

Оператор **DATA** имеет вид

DATA : <имя массива> {, <константа>} ... ;

Имя массива – идентификатор.

Пример.

```
CONST: G, GAMMA, 1, NGCOM;  
ZONE: 1, 300., FE, .0847;  
ZONE: 2, 300., H, .067, 0., .0335;  
DATA: NGCOM, 0,0,0,0,0,0,1,1,2,2,2,3,3,0,0;
```

Этот фрагмент задания требует рассчитать γ -константы с помощью программы GAMMA (вариант программы АРМАКО-G). Четвертый параметр оператора CONST – NGCOM – является именем массива, содержащего описание свертки рассчитанных макроконстант по группам энергий γ -квантов. Этот массив задается оператором DATA.

Оператор FORM предназначен для перевода массива констант из одного формата в другой. Под форматом в данном случае понимается структура массива констант, т.е. порядок следования входящих в него значений физических величин. Перевод массива из одного формата в другой означает изменение порядка следования значений величин, может быть, перевычисление некоторых из значений.

Оператор FORM имеет вид

FORM: <имя массива>, <формат>, <имя массива> ;

Все три параметра – идентификаторы. Первый из них – имя массива, который вырабатывается оператором FORM. Второй – имя формата, в который переводится массив (исходный формат задавать не требуется, так как он является атрибутом исходного массива). Третий параметр – имя исходного массива.

Преобразование форматов выполняется специальными программами – преобразователями.

Оператор TAPE – неисполняемый оператор. Он предназначен для указания места во внешней памяти, куда должен быть помещен массив констант с данным именем. Этот массив вырабатывается одним из исполняемых операторов (CONST или FORM). Запись массива производится во время исполнения этих операторов.

Оператор TAPE имеет вид

TAPE: <имя массива>, <адрес>, <начало>, <длина>;

Здесь <адрес> – либо номер канала (в этом случае последние два параметра не имеют смысла), либо математический номер внешнего устройства; <начало> – номер зоны, начиная с которой будет размещен массив констант, а <длина> – количество зон, отведенных на устройстве для

хранения массива. Все три рассмотренных параметра задаются в виде целых десятичных констант.

Оператор *END* – это оператор конца задания.

6.2.3. Общая схема функционирования. Пакет ОКС предназначен для проведения расчетов групповых макроконстант на машине БЭСМ-6 в среде операционной системы Диспак и мониторной системы Дубна.

Задание для пакета помещается вместе с другими начальными данными к задаче расчета полей излучения, для решения которой необходимы макроконстанты, получаемые в соответствии с этим заданием. Запуск пакета ОКС происходит во время выполнения программы расчета полей излучения и осуществляется с помощью операторов

$$\left\{ \begin{array}{l} \text{CALL} \\ \text{CALL LOADGO} \end{array} \right\} \quad \text{ОКС} \langle \text{список параметров} \rangle$$

В качестве параметров указываются начала и размеры областей оперативной памяти, в которых пакет может размещать исходный текст и внутреннее представление задания.

К ОКС можно обратиться и отдельным запуском с помощью управляющего предложения *MAIN ОКС1.

На ресурсах внешней памяти пакета ОКС постоянно находится следующая информация:

- библиотека паспортов операторов; паспорт оператора помимо информации, описывающей этот оператор, содержит ссылку на процедуру-адаптер; последняя обеспечивает интерфейс между пакетом ОКС и соответствующей константной системой;
- библиотека объектных модулей процедур-адаптеров и других процедур системы ОКС;
- библиотека объектных модулей константных программ;
- библиотеки микроконстант.

Основные программные компоненты системного наполнения пакета выполняют следующие функции.

Анализатор вводит исходный текст задания и с помощью процедуры синтаксического разбора переводит его на внутренний язык.

Интерпретатор, просматривая внутреннее представление задания, последовательно реализует с помощью процедур-адаптеров все исполняемые операторы.

Вызов константных программ выполняется с помощью специально разработанной процедуры LOADM. Эта процедура запоминает содержимое оперативной памяти, регистров и

некоторую другую информацию на ресурсах внешней памяти, освобождая тем самым всю оперативную память для вызываемой программы. По окончании работы этой программы (при выполнении оператора RETURN) восстанавливается все необходимое для продолжения работы ОКС.

Опыт разработки и эксплуатации пакета ОКС показал, что даже относительно скромная, но целенаправленная системная поддержка набора вспомогательных прикладных программ позволяет достаточно эффективно организовать использование этих программ в режиме «черного ящика», т.е. не затрудняя пользователя необходимостью освоения их специфики. Процессорное время, затрачиваемое системным наполнением пакета ОКС, составляет от 2% до 5% процессорного времени выполнения расчета констант в целом.

6.2.4. Пример работы с пакетом ОКС

```
CONST: Y, NEUTR, 26, 2, 2, 1;  
ZONE: 1, 595., H, .452,, 0, .266,,  
      B-10, 7.55E-9;  
ZONE: 2, 595., Fe, .848;  
FORM: A, ROZ-6, Y;  
TAPE: A, 5;  
END;
```

В данном примере задается расчет нейтронных констант для двух физических зон. Полученные константы требуется перевести в формат ROZ-6 и записать в канал с номером 5 (математический номер 45).

Первый оператор означает обращение к константной программе NEUTR. Y – имя полученного массива макроконстант. 26 – число групп. 2, 2, 1 – параметры, определяющие работу программы.

Два следующих оператора описывают физические зоны – их температуры и состав. Эта информация является исходной для программы NEUTR.

Оператор FORM означает перевод констант из формата FMAC5A, в котором они были получены, в формат ROZ-6. A – имя массива, являющегося результатом.

Оператор TAPE задает номер канала, в который должен быть помещен массив A.

END – оператор конца задания.

6.3. Пакет МП: автоматическое планирование вычислений

6.3.1. Архитектура. Автоматическое планирование вычислений, т.е. автоматическая генерация расчетных цепочек модулей является одним из средств повышения уровня тематической квалификации пакета. Прежде всего это выражается в том, что пользователь пакета, осуществляющего автоматическое планирование вычислений (такие пакеты иногда называют интеллектуальными или думающими пакетами), имеет возможность при составлении задания описывать не процесс решения задачи, а только ее постановку, т.е. исходную предметную ситуацию, известные и искомые в этой ситуации данные. Ясно, что эта возможность – непроецедурный язык заданий и практически полное экранирование от пользователя механизмов генерации расчетной программы – делает интеллектуальный пакет весьма привлекательным для широкого круга конечных пользователей. Следует отметить, что средства автоматического планирования вычислений особенно полезны и эффективны в том случае, когда пакет ориентирован на обслуживание расчетов, программы для которых репродуктивно конструируются из функциональных модулей расчета стереотипных предметных конфигураций (например, таких, как консольная балка, кинематическая пара, электрический фильтр напряжения или тока и т.п.).

Как уже отмечалось в гл.1, специфика того или иного подхода к организации автоматического планирования вычислений проявляется главным образом, во-первых, в методе планирования, реализуемом планировщиком пакета, и, во-вторых, в содержании и форме представления знаний о предметной области и функциональном наполнении пакета, которые могут быть накоплены в подсистеме внутреннего информационного обслуживания и использованы при составлении расчетной цепочки.

Широкое практическое применение получил подход, при котором автоматическое составление расчетной цепочки сводится к поиску произвольного транзитивного замыкания для известных (или искомых) величин задачи на специальном графе, называемом вычислительной моделью предметной области (задачи) [58,86,89]. Вычислительная модель содержит знания, необходимые для составления плана вычислений в виде описания информационных интерфейсов между модулями функционального наполнения. Причем отличительной особенностью такого описания является то, что информационные интерфейсы задаются через априорно установленные имена величин предметной области и только эти имена

используются в качестве фактических параметров модулей, включаемых планировщиком в расчетную цепочку.

В рамках исследований по пакетной проблематике, проводимых в ИПМ имени М.В.Келдыша АН СССР, рассматривался один из возможных подходов к организации автоматического планирования вычислений, позволяющий при не определенных заранее информационных связях между модулями пакета строить не только правильную, но и достаточно близкую к оптимальной расчетную цепочку с учетом различных критериев оптимизации, а также эффектов вычисления нескольких величин одним модулем. В основе этого подхода лежит использование в процессе планирования вычислений разнообразных знаний как о предметной области, охватываемой пакетом, так и о семантике функциональных модулей. В результате был реализован методический пакет МП [71,94,139].

В качестве предметной области пакета МП выбран класс задач вычислительной геометрии, которые могут быть поставлены на чертеже, рассматриваемом как совокупность точек, отрезков, углов, треугольников и четырехугольников и решение которых не требует дополнительных построений. На выбор этого класса задач оказали влияние следующие мотивировки:

- наглядность формулировок геометрических задач;
- возможность постановки задач, решение которых реализуется довольно длинными расчетными цепочками, состоящими из 10-15 модулей;
- и, главное, большое число (даже для относительно простого чертежа) возможных конкретных конфигураций, к которым потенциально применим модуль, что делает затруднительным для пользователя априорное перечисление всех возможных конкретных наборов фактических параметров модуля.

Последнюю мотивировку можно проиллюстрировать следующим примером. На рис.6.2 имеется 14 треугольников. Модулю, вычисляющему третью сторону треугольника по двум сторонам и углу между ними, в каждом треугольнике могут соответствовать 3 набора фактических параметров (так как с точностью до перестановки внутри пары сторон имеются три различные пары). Следовательно, только для данного модуля на этом относительно простом чертеже имеется 42 различных набора фактических параметров.

Что же касается свойств системного обеспечения пакета МП, то их определение обусловлено следующими соображениями, отражающими, с одной стороны, ориентацию на разра-

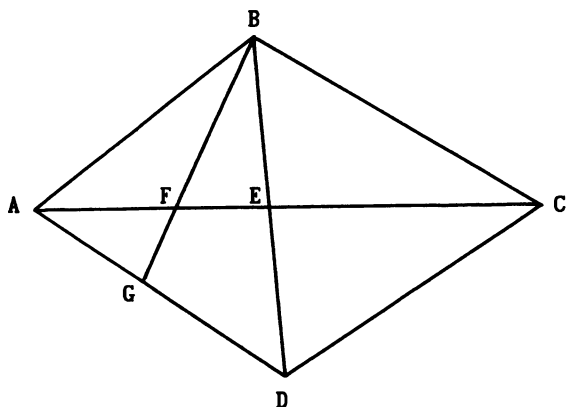


Рис 6.2. 14 треугольников

ботку интеллектуального пакета для конкретной предметной области, а с другой, – попытку исследовать в ходе этой разработки общие вопросы организации и функционирования таких пакетов.

Естественно предположить, что при работе с интеллектуальным пакетом пользователь предпочтет следовать такой же дисциплине проведения расчетов, какой он придерживался при традиционном (ручном) решении задач. В связи с этим конструкции языка заданий пакета должны быть прежде всего лексически и синтаксически ориентированы на наглядное и компактное формулирование следующих трех основных фаз расчета:

- описание предметной ситуации (в данном случае чертежа);
- постановка задачи, т.е. указание известных и искоемых элементов ситуации, а также других условий решения задачи;
- запрос на решение задачи при конкретных числовых значениях известных элементов ситуации.

Вообще говоря, чертеж, на котором будет поставлена задача, может быть задан в терминах любых из перечисленных выше геометрических конфигураций (элементов). Однако такая свобода описания чертежа требует значительных и вряд ли оправданных затрат (по времени и по памяти) при проведении анализа заданного чертежа. Поэтому было реше-

но задавать описание чертежа в виде перечисления всех представленных на нем прямых и принадлежащих им точек.

При выборе регламента модуляризации функционального наполнения интеллектуального пакета приходится учитывать два в известной степени противоречивых обстоятельства. С одной стороны, желательно ввести некоторую жесткую дисциплину программирования модулей, которая обеспечивала бы их языковую и операционную однородность. Это позволило бы снизить системные издержки на организацию межмодульного интерфейса и использовать при генерации и исполнении расчетных программ достаточно ограниченный набор штатных и специальных системных средств. С другой стороны, для выполнения эффективного автоматического планирования расчетных цепочек требуется высокая контекстная независимость модулей, и поэтому желательно оформлять модули инвариантно относительно возможных информационных связей с какими-либо другими модулями. Сбалансированное разрешение этих противоречий может быть найдено на основе использования регламента смешанной модуляризации. При таком подходе нормативы «внутренней» составляющей регламента модуляризации обеспечивают единообразие в подготовке программных тел модулей, а нормативы «внешней» составляющей позволяют описывать в заголовках модулей их семантику и, в частности, семантику входных и выходных величин модуля.

Принципиальная схема автоматического синтеза расчетных программ, принятая в качестве основы пакета МП, выглядит следующим образом. Знания о предметной области и функциональных модулях хранятся в форме семантической сети. Знания о предметной области описывают свойства абстрактных объектов предметной области и отношения между ними. Эти знания используются при интерпретации описания чертежа, на котором будет поставлена задача. Знания о функциональных модулях, помимо программ вычислительных процедур, содержат сведения об их семантике: абстрактные описания геометрических конфигураций, к которым применим данный модуль, а также описание, характеризующее саму программную реализацию вычислительной процедуры. Сведения о семантике модуля используются при планировании, во-первых, для определения на заданном чертеже тех конкретных геометрических конфигураций, к которым применим данный модуль, и, во-вторых, при вычислении цены модуля по тем его характеристикам, перечень которых определяется заданным в постановке задачи критерием оптимизации. Собственно планирование понимается как

построение минимальной по цене расчетной цепочки, последовательное выполнение модулей которой при заданных значениях величин, предполагаемых при постановке задачи известными, приводит к вычислению значений величин, указанных в постановке задачи в качестве искомых. При этом цена расчетной цепочки определяется как сумма цен входящих в нее модулей. По сформированной планировщиком расчетной цепочке генерируется программа, реализующая решение поставленной задачи.

При определении операционных возможностей пакета МП было учтено, что наиболее удобным для пользователя режимом общения с пакетом при проведении расчетов в рамках протокола «ситуация – задача – решение» является режим диалога. Для сокращения времени отклика пакета на запрос, касающийся решения задачи, обработка постановки задачи завершается компиляцией расчетной программы на языке программирования функциональных модулей. Кроме того, для упрощения инфраструктуры системного наполнения пакета было решено воспользоваться единой базой данных для хранения знаний о предметной области и функциональных модулях, а также расчетных данных.

Положенная в основу пакета МП схема автоматической генерации расчетных программ требует включения в состав системного обеспечения пакета развитых средств информационного обслуживания. Для эффективной обработки разнообразных информационных запросов, возникающих в связи с синтезом расчетной программы, эти средства должны осуществлять управление обширной базой данных и обеспечивать при этом возможность логического анализа знаний о предметной области и функциональных модулях, сведений о конкретной предметной ситуации и поставленной на ней задаче. В аспекте внешнего обслуживания информационные средства интеллектуального пакета должны обеспечивать возможность ввода в базу данных сведений о предметной области и функциональных модулях (в том числе текстов вычислительных процедур) и, разумеется, выдачу этих сведений в ответ на соответствующие запросы.

Анализ изложенных выше соображений позволил сделать вывод о целесообразности разработки пакета МП в виде надстройки над многоцелевой информационно-логической системой. Поскольку к моменту начала работ над пакетом с автоматическим планированием вычислений имелся значительный опыт использования информационно-логической системы ВОПРОС-ОТВЕТ-2 [92], она и была выбрана в качестве

организационной основы пакета МП. Рассмотрим теперь особенности реализации этого пакета.

6.3.2. Представление знаний. Использование такой системы, как ВОПРОС-ОТВЕТ-2, дает возможность накапливать знания в трех видах: в виде фактов, правил вывода и в виде процедур.

Знания о предметной области, содержащиеся в базе данных пакета МП, представляются в виде набора фактов и правил вывода, в частности, правил, определяющих взаимосвязь между объектами предметной области. Например, частью описания конкретного геометрического чертежа могут быть сведения о перпендикулярности некоторых отрезков, и одновременно в базе данных может присутствовать правило вывода, утверждающее, что если две стороны одного треугольника взаимно перпендикулярны, то такой треугольник является прямоугольным.

Знания о вычислениях, возможных в данной предметной области, накапливаются в базе данных в виде модулей функционального наполнения. Регламент модуляризации, принятый в пакете МП, предусматривает, что функциональный модуль состоит из двух частей: тела – программы, записанной в терминах вычислительных операторов языка системы ВОПРОС-ОТВЕТ-2, и заголовка – набора фактов и правил вывода, описывающих семантику модуля. По сути дела, заголовки модулей представляют собой записанные в виде фактов и правил вывода знания о процедурных знаниях, накопленных в пакете. В свою очередь, заголовок модуля также состоит из двух частей: описания проблемной семантики модуля и описания его системной семантики.

Проблемная семантика модуля содержит сведения о том, как этот модуль может быть использован при расчетах, проводимых в рамках рассматриваемой предметной области. Прежде всего, проблемная семантика содержит сведения о геометрической конфигурации, к которой данный модуль применим. Например, такой конфигурацией может быть прямоугольник или отрезок, которому принадлежит некоторая точка, и т.д. Кроме того, проблемная семантика модуля содержит сведения о том, какие элементы данной конфигурации могут рассматриваться как входные или выходные величины этого модуля. Так, если конфигурация представляет собой произвольный треугольник, то в описании проблемной семантики одного из модулей, применимых к данной конфигурации (а именно модуля, реализующего фор-

малу Герона), входными величинами объявляются три стороны треугольника, а выходной величиной – его площадь.

Вообще говоря, один и тот же модуль способен обчислять разные конфигурации, может быть, даже относящиеся к разным предметным областям. Поэтому проблемная семантика модуля может содержать сведения о нескольких конфигурациях, к которым этот модуль в принципе применим.

Системная семантика модуля содержит сведения о модуле как о процедуре, т.е. сведения о различных программных параметрах и эксплуатационных характеристиках. Это, например, адрес начала программы, список идентификаторов соответствующих входных и выходных величин, размерность этих величин, количество команд, время счета и т.д. Характер этих знаний заранее не фиксирован, что позволяет накапливать в базе данных любую информацию, которая может быть полезна при планировании.

Наличие в базе данных знаний о предметной области и семантике модулей дает возможность решать задачу организации автоматического планирования вычислений в более полном объеме, чем это делается при традиционной схеме. Действительно, во-первых, используя описания системной семантики модулей, можно производить не просто планирование вычислений, а оптимальное планирование, причем с учетом различных критериев оптимизации. В частности, выбирая соответствующую информацию из базы данных, планировщик может построить цепочку вычислений, используя такие функциональные модули, которые дают минимальное суммарное время их выполнения или занимают минимальный по размеру участок памяти. Критерий выбора модулей может быть и более сложным, например: минимальное время при заданных ограничениях на память. Разумеется, при подборе модулей могут учитываться не только такие параметры, как время и используемая память, но и другие самые разнообразные свойства модулей.

С другой стороны, описания проблемной семантики модулей обеспечивают универсальность их применимости. Как уже отмечалось, при планировании вычислений на основе вычислительных моделей модули, образующие расчетную цепочку, сцепляются по наименованиям их входов и выходов. Существенным недостатком такого способа планирования является необходимость заранее предусмотреть все возможные сцепления модулей, которые будут возникать в цепочках, строящихся при планировании конкретных расчетов. В самом деле, предвидеть, в каких контекстах понадобится использовать данный модуль и, следовательно, какие вели-

чины могут соответствовать его входам и выходам, далеко не всегда удастся даже при решении сравнительно простых задач. Так, например, при решении геометрических задач часто бывает необходимо использовать модуль, способный обчислять прямоугольные треугольники, но не всегда заранее можно установить, что некоторый треугольник прямоугольный. Наличие же в базе данных пакета МП знаний о предметной области и абстрактного описания обчисляемого данным модулем конфигурации позволяет непосредственно в процессе планирования выявить все те конфигурации, к которым рассматриваемый модуль применим.

Заметим также, что знания о предметной области и знания о возможных в ней вычислениях могут накапливаться и корректироваться независимо друг от друга. Возникающие при этом терминологические различия могут быть ликвидированы посредством ввода в базу данных пакета правил вывода, определяющих синонимию. Разделение сложной задачи накопления знаний на части, каждая из которых может решаться независимо, позволяет применять эффективную технологию создания пакета.

6.3.3. Язык пакета МП. В связи с тем, что пакет МП разрабатывался в виде надстройки над информационно-логической системой ВОПРОС-ОТВЕТ-2, его язык был определен как расширение языка этой системы. Языковые конструкции системы ВОПРОС-ОТВЕТ-2 используются для представления знаний о предметной области, семантики модулей и, наконец, для программирования этих модулей. Система позволяет разработчику пакета в режиме диалога вводить в базу данных новые знания и уточнять уже имеющиеся, используя при этом лексику естественного языка. Пользователь, решающий геометрические задачи, также работает с пакетом МП в режиме диалога, применяя при этом специальные операторы языка пакета, которые позволяют в удобной форме задавать основные фазы расчета. Рассмотрим эти операторы более подробно.

Для описания чертежа используется оператор

СИТУАЦИЯ: P_1, P_2, \dots, P_n

Здесь P_i ($1 \leq i \leq n$) – либо триплет «характеристика – объект – значение», либо конструкция вида «ПРЯМАЯ – $T_1 T_2 \dots T_k$ », где T_j ($1 \leq j \leq k$) – буквы, обозначающие на чертеже точки, принадлежащие данной прямой. Эти буквы должны быть перечислены в порядке их следования в

произвольном направлении. Для полного описания чертежа необходимо указать все представленные на нем прямые и все принадлежащие им точки. Кроме того, нужно с помощью триплетов описать существенные отношения, имеющие место в пределах рассматриваемой ситуации, такие, например, как перпендикулярность отрезков, фиксированный размер каких-то элементов или их равенство и т.д. Программа, интерпретирующая этот оператор, используя знания о предметной области, содержащиеся в базе данных, порождает все элементы данной ситуации и в конечном счете заносит в базу данных подробное описание их свойств, хотя ни сами элементы, ни их свойства прямо в операторе не указываются. Элементами ситуации, кроме точек, могут быть отрезки, углы, треугольники, четырехугольники и некоторые другие объекты. В качестве индивидуальных имен этих объектов используются принятые в геометрии обозначения, составленные из букв, идентифицирующих точки. Например, отрезок – АВ, угол – У-АВС, треугольник АСD. Порядок букв в таких составных именах при порождении имен интерпретирующей программой устанавливается следующим образом: в именах отрезков и треугольников буквы следуют в алфавитном порядке, в именах углов на втором месте стоит имя вершины, а на первом и третьем – имена концов в алфавитном порядке, в именах четырехугольников на первом месте первая в алфавитном порядке вершина, на втором – первая в алфавитном порядке из двух смежных с ней и далее в полученном направлении обхода. Отметим, что если пользователь употребляет имена элементов ситуации в операторах системы ВОПРОС-ОТВЕТ-2, то он должен сохранять указанный способ их именования. Что же касается рассматриваемых ниже операторов языка пакета МП, то в них порядок букв может быть произвольным. При интерпретации операторов буквы будут автоматически представляться нужным образом.

Постановка конкретной задачи осуществляется с помощью оператора, имеющего вид:

ЗАДАЧА: ДАНО A_1 И A_2 И ... И A_n ,
НАЙТИ B_1 И B_2 И ... И B_m ,
ОПТИМИЗИРУЯ C_1 И C_2 И ... C_k , **НАЧАЛО ПРОГРАММЫ** N

Здесь A_i ($1 \leq i \leq n$) – имена элементов ситуации, объявленных заданными, B_j ($1 \leq j \leq m$) – имена элементов, которые требуется найти, C_l ($1 \leq l \leq k$) – параметры оптимизации, а N – адрес, начиная с которого должна быть

размещена реализующая решение данной задачи программа. Список заданных элементов, вообще говоря, может отсутствовать, поскольку необходимые исходные величины могут быть уже определены при описании ситуации. Параметры оптимизации (такие, как время, длина и т.д.) являются характеристиками триплетов, входящих в описание модулей. Значения этих характеристик представляют собой числа, определяющие конкретные величины параметров. Кроме триплетов, задающих значения параметров оптимизации, в базе данных должны содержаться триплеты, определяющие вес соответствующих параметров. Истинная цена модуля в контексте решаемой задачи представляет собой сумму произведений весов параметров оптимизации, перечисленных в рассматриваемом операторе, на конкретные значения этих параметров, указанные в описаниях семантики соответствующих модулей.

Среди параметров оптимизации есть особый параметр — *цена*. Если указание об оптимизации отсутствует, то оптимизация идет по этому параметру.

В результате выполнения оператора ЗАДАЧА генерируется готовая программа, обеспечивающая заданный расчет с учетом критериев оптимизации, и на терминал выдается сообщение о расположении программы в памяти. Если задача не может быть решена, то на терминал также выдается соответствующее сообщение. В заключение, отметим, что на одном и том же чертеже можно задавать и получать совершенно различные элементы, решая, таким образом, разные задачи, которые ставятся в рамках одной ситуации.

Запуск полученной программы на счет осуществляется с помощью оператора

**ЗАПУСК: ПРИ $A_1=K_1$ И $A_2=K_2$ И . . . И $A_n=K_n$,
НАЧАЛО ПРОГРАММЫ N**

Здесь A_i и K_i ($1 \leq i \leq n$) — соответственно имена исходных элементов ситуации и задаваемые начальные значения этих элементов, а N — адрес начала построенной ранее программы. В результате выполнения рассматриваемого оператора вычисляются искомые величины и полученные значения выдаются на терминал. Разумеется, одна и та же программа может просчитываться несколько раз с разными начальными данными. В том случае, когда значения каких-либо исходных элементов не определены, на терминал выдается соответствующий отказ.

6.3.4. Общая схема функционирования. В общем виде пакет МП представляется состоящим из четырех компонентов (рис.6.3): системы ВОПРОС-ОТВЕТ-2 и надстроенных над ней исполнителей операторов СИТУАЦИЯ, ЗАДАЧА и ЗАПУСК.

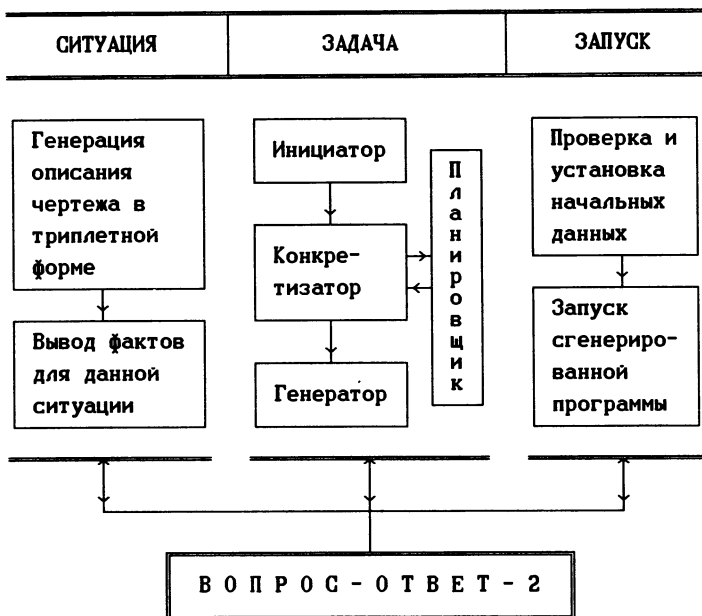


Рис.6.3 Структура пакета МП

Программы исполнителей, как и системы ВОПРОС-ОТВЕТ-2, реализованы на автокоде БЕМШ.

Как уже отмечалось, в результате выполнения оператора описания ситуации в базу данных заносится информация о свойствах элементов ситуации и отношениях между ними. Сначала исполнитель СИТУАЦИЯ, используя указанные в обрабатываемом операторе прямые и принадлежащие им точки, порождает триплеты, которые описывают все имеющиеся на данном чертеже отрезки. Затем из отрезков конструируются углы, треугольники и четырехугольники. Свойства и отношения полученных элементов, определяемые структурой чертежа, дополняются фактами, представленными триплетами, которые являются параметрами обрабатываемого опера-

тора. Вся эта информация непосредственно заносится в базу данных и используется второй частью исполнителя СИТУАЦИЯ при выводе с помощью правил, представляющих общие знания о геометрии, более подробных сведений о чертеже.

Исполнитель оператора ЗАДАЧА работает следующим образом. Прежде всего осуществляется перевод постановки задачи во внутреннее представление. Во внутреннем представлении каждому элементу ситуации, входящему в множество известных или искомых, ставится в соответствие некий номер. Далее при планировании номера приписываются также элементам ситуации, оказывающимся промежуточными величинами расчета. Для хранения значений элементов ситуации, участвующих в расчете, отведен фрагмент массива Э, начиная с Э(1001), причем элементу с номером N соответствует компонент массива Э с номером $1001+N$. Планирование расчета рассматривается как задача поиска минимального пути в пространстве состояний. Состояния соответствуют всевозможным подмножествам множества входных и выходных величин всех функциональных модулей. Переход из состояния в состояние осуществляется применением одного из модулей. Во внутреннем представлении любое состояние задается битовой строкой. Если некий элемент ситуации с внутренним номером N входит в данное состояние (т.е. если его значение считается в этом состоянии известным), то в соответствующем разряде внутреннего представления стоит 1. Множества входов и выходов модуля M после сопоставления с некоторой конкретной конфигурацией элементов ситуации также задаются парой битовых строк, обозначаемых соответственно $vx(M)$ и $вых(M)$. Такое внутреннее представление состояний, а также входов и выходов модулей удобно для реализации операций, используемых при планировании.

Планирование расчетной цепочки осуществляется двумя работающими во взаимодействии компонентами исполнителя ЗАДАЧА: собственно планировщиком и конкретизатором, извлекающим необходимую для планирования информацию из базы данных и переводящим ее во внутреннее представление. Планировщик работает исходя из следующих предпосылок. В данной ситуации заданы начальное и целевое состояние, т.е. множество элементов ситуации, значения которых предполагаются известными, и множество элементов, значения которых нужно вычислить. Требуется найти минимальную по цене цепочку модулей из функционального наполнения пакета, последовательное применение которых к начальному

состоянию позволяет определить значения элементов целевого состояния. Модуль M считается применимым к состоянию S , если $vx(M) \subseteq S$. Применение модуля M дает новое состояние $S' \supseteq vx(M)$.

Специфика планирования вычислений по сравнению с планированием произвольных направленных действий заключается в том, что $S' = S \cup vx(M)$. Поэтому при планировании исключается (при существовании решения) возможность построения тупикового пути.

В планировщике исполнителя ЗАДАЧА применен простой метод планирования, заключающийся в построении нескольких случайно подобранных, но очищенных от лишних вычислений цепочек и в выборе из них в качестве решения цепочки с минимальной ценой. Тот факт, что при построении случайной цепочки невозможны тупики, а также то, что из цепочек исключаются лишние модули, чем обеспечивается приемлемое качество любой получаемой цепочки, делает предложенный способ планирования достаточно практичным и эффективным. Число цепочек определяется ресурсом, представляющим собой ограничение на суммарную длину сгенерированных цепочек. Время практически линейно зависит от выбранного ресурса, и если он достаточно велик, то результатом планирования будет цепочка, близкая к минимальной.

Генерация случайной цепочки происходит следующим образом. Планировщик в ответ на запрос о модуле получает от конкретизатора описание во внутреннем представлении входов и выходов некоего случайно выбранного модуля — $vx(M)$ и $vyx(M)$. Если для текущего состояния (в качестве первого текущего состояния берется начальное) выполняются условия $S \supseteq vx(M)$ и $\neg(S \supseteq vyx(M))$, т.е. если модуль M применим к S и его применение позволяет вычислить значения новых элементов ситуации, то модуль M включается в планируемую цепочку, а в качестве текущего берется состояние $S' = S \cup vyx(M)$. Если эти условия не выполняются, то следует запрос к конкретизатору с требованием предоставить описание другого модуля. Построение цепочки успешно завершается, если для некоторого текущего состояния окажется, что целевое состояние является его подмножеством. Далее обратным просмотром из полученной цепочки исключаются лишние модули, т.е. модули, не участвующие в вычислении значений элементов целевого состояния. После этого планировщик генерирует запросы к конкретизатору о цене модулей, входящих в

«чистую» цепочку, и, суммируя полученные значения, определяет ее цену.

Как видно из описания функций планировщика, основными задачами конкретизатора являются извлечение необходимой для планировщика информации о конкретных конфигурациях, обсчитываемых модулями, и о значениях тех характеристик модулей, которые являются параметрами оптимизации, а также перевод этой информации во внутреннее представление. Работа конкретизатора представляет собой диалог с системой ВОПРОС-ОТВЕТ-2, в котором конкретизатор генерирует вопрос, обрабатывает множество полученных ответов, по результатам обработки снова генерирует вопрос к системе и т.д. Конкретизатор работает следующим образом. Первоначально он формирует внутреннее представление начального и целевого состояний и передает эту информацию планировщику. В ответ на запрос планировщика о модуле конкретизатор случайным образом выбирает модуль из имеющегося у него списка модулей и формирует запрос к базе данных об обсчитываемых этим модулем конфигурациях. Возможно, что этот модуль в заданном состоянии не применим или его применение не вырабатывает новых известных величин. Тогда, в ответ на повторные запросы планировщика, конкретизатор осуществляет полный перебор (по кольцу) всего списка модулей, начиная с данного. Если ни один модуль не подходит, значит задача не имеет решения. Если же в некотором состоянии какой-либо модуль применим несколькими способами (например, когда имеется несколько треугольников, в которых известно по две стороны и одному углу), то конкретизатор выбирает случайное возможное применение и выдает его планировщику. Помимо этого, конкретизатор в ответ на соответствующий запрос планировщика вычисляет цену модуля, извлекая значения его характеристик из базы данных.

Последним этапом интерпретации оператора ЗАДАЧА является генерация текста программы, реализующей спланированный расчет. Выполнение этого этапа не представляет принципиальной трудности, поскольку конкретизатором были получены все необходимые данные: соответствия между элементами ситуации и входными и выходными ячейками модулей результирующей цепочки, номера начальных операторов модулей, являющиеся их идентификаторами в системе ВОПРОС-ОТВЕТ-2, а также значения элементов ситуации, определенные при ее задании. Вместе со спланированной цепочкой эти данные однозначно определяют последовательность вызовов модулей и интерфейс между ними. Сгенериро-

ванная программа помещается на хранение в базу данных. Идентификатором программы служит номер ее первого оператора, задаваемый при постановке задачи. Вместе с программой запоминаются данные, необходимые для ее запуска, а также (для проверки корректности запуска) подмножество элементов начального состояния, которое использовалось в найденном решении.

Работа исполнителя ЗАПУСК начинается с проверки полноты множества значений тех элементов начального состояния, которые используются при найденном решении. В том случае, если значения некоторых из этих элементов не определены, на терминал выдается соответствующий отказ. Если же при проверке полнота подтвердилась, исполнитель ЗАПУСК формирует и передает на обработку системе ВОПРОС-ОТВЕТ-2 вычислительный оператор вида ВЫПОЛНИ <номер оператора>, который обеспечивает передачу управления на первый оператор сгенерированной программы расчета. После выполнения этой программы значения элементов целевого состояния выдаются на терминал.

В следующем разделе приводится протокол работы с пакетом МП, снабженный соответствующими комментариями.

6.3.5. Протокол работы с пакетом МП. Далее следует протокол сеанса работы с пакетом МП. Основной комментарий к протоколу будет дан в конце раздела. Здесь отметим только, что сообщения, выдаваемые пакетом, печатаются в протоколе с отступом в четыре позиции.

НАЧИНАЙТЕ РАБОТУ
ПРЕДПИСАНИЕ 'ЧТЕ СЕМАНТИКА'
ПОНЯТНО

ПРОЧТИ ГЕОМЕТРИЮ

СИТУАЦИЯ: ПРЯМАЯ - А В, ПРЯМАЯ - В С, ПРЯМАЯ - С D,
ПРЯМАЯ - А G D, ПРЯМАЯ - А F E C, ПРЯМАЯ - В E D,
ПРЯМАЯ - В F G, РАВНЫ AF FC, РАВНЫ BE ED

ЗАДАЧА: ДАНО АВ И АF И FE И У-ВАF, НАЙТИ S-CDE,
ОПТИМИЗИРУЯ ВРЕМЯ, НАЧАЛО ПРОГРАММЫ 10

ЗАДАЧА РЕШЕНА, ПРОГРАММА С 10 ДО 94

ВЫДАЙ ПРОГРАММУ С 10 ДО 94

10 ПУСТЬ P(1) = Э(1001)

12 ПУСТЬ P(2) = Э(1002)

14 ПУСТЬ P(3) = Э(1004)

16 ВЫПОЛНИ 1000

18 ПУСТЬ Э(1016) = P(4)

20 ПУСТЬ Э(1021) = P(6)

22 ПУСТЬ $Z(1047) = P(5)$
 24 ПУСТЬ $P(1) = Z(1021)$
 26 ВЫПОЛНИ 1500
 28 ПУСТЬ $Z(1022) = P(2)$
 30 ПУСТЬ $P(1) = Z(1002)$
 32 ВЫПОЛНИ 1233
 34 ПУСТЬ $Z(1006) = P(2)$
 36 ПУСТЬ $P(2) = Z(1003)$
 38 ПУСТЬ $P(1) = Z(1006)$
 40 ВЫПОЛНИ 1400
 42 ПУСТЬ $Z(1007) = P(3)$
 44 ПУСТЬ $P(1) = Z(1003)$
 46 ПУСТЬ $P(2) = Z(1016)$
 48 ПУСТЬ $P(3) = Z(1022)$
 50 ВЫПОЛНИ 1000
 52 ПУСТЬ $Z(1013) = P(4)$
 54 ПУСТЬ $Z(1055) = P(6)$
 56 ПУСТЬ $Z(1025) = P(5)$
 58 ПУСТЬ $P(1) = Z(1013)$
 60 ВЫПОЛНИ 1233
 62 ПУСТЬ $Z(1015) = P(2)$
 64 ПУСТЬ $P(1) = Z(1025)$
 66 ВЫПОЛНИ 1233
 68 ПУСТЬ $Z(1028) = P(2)$
 70 ПУСТЬ $P(1) = Z(1007)$
 72 ПУСТЬ $P(2) = Z(1015)$
 74 ПУСТЬ $P(3) = Z(1028)$
 76 ВЫПОЛНИ 1000
 78 ПУСТЬ $Z(1029) = P(4)$
 80 ПУСТЬ $Z(1053) = P(6)$
 82 ПУСТЬ $Z(1046) = P(5)$
 84 ПУСТЬ $P(1) = Z(1007)$
 86 ПУСТЬ $P(2) = Z(1015)$
 88 ПУСТЬ $P(3) = Z(1029)$
 90 ВЫПОЛНИ 1600
 92 ПУСТЬ $Z(1005) = P(4)$
 94 ВОЗВРАТ

ВЫЧИСЛИ $A = \text{ARCSIN } 0.6$

ВЫДАИ A

.643501108

ЗАПУСК: ПРИ $AB = 5$ И $AF = 4.5$ И $FE = 2.3$ И $Y-FAB = 0.78$,
 НАЧАЛО ПРОГРАММЫ 10

S-CED = .298597720E1

ЗАПУСК: ПРИ $BA = 8.7$ И $AF = 4.5$ И $FE = 2.3$ И
 $Y-FAB = 0.78$,

НАЧАЛО ПРОГРАММЫ 10

S-CED = .673038404E1

ВОПРОС 'НАЧАЛО K1 1000'

НАЧАЛО АО 1000

ВОПРОС 'МОДУЛЬ АО K1'

МОДУЛЬ АО ПО-ДВУМ-СТОРОНАМ-ТРЕУГОЛЬНИКА-И-УГЛУ-
МЕЖДУ-НИМИ-ВЫЧИСЛЯЕТ-ОСТАЛЬНЫЕ-УГЛЫ-И-ТРЕТЬЮ-
СТОРОНУ, ДЛИНА-ПРОГРАММЫ-16

ВЫДАЙ ПРОГРАММУ С 1000 16

1000 ВЫЧИСЛИ $P(4) = P(1) P(1)$

1001 ВЫЧИСЛИ $T = P(2) P(2)$

1002 ВЫЧИСЛИ $P(4) = P(4) + T$

1003 ВЫЧИСЛИ $T = \cos P(3)$

1004 ВЫЧИСЛИ $T = T^2$

1005 ВЫЧИСЛИ $T = T P(1)$

1006 ВЫЧИСЛИ $T = T P(2)$

1007 ВЫЧИСЛИ $P(4) = P(4) - T$

1008 ВЫЧИСЛИ $P(4) = \sqrt{P(4)}$

1009 ВЫЧИСЛИ $P(5) = \sin P(3)$

1010 ВЫЧИСЛИ $P(5) = P(5) P(2)$

1011 ВЫЧИСЛИ $P(5) = P(5) : P(4)$

1012 ВЫЧИСЛИ $P(5) = \arcsin P(5)$

1013 ВЫЧИСЛИ $P(6) = 3.142 - P(5)$

1014 ВЫЧИСЛИ $P(6) = P(6) - P(3)$

1015 ВОЗВРАТ

ВОПРОС 'ВРЕМЯ АО K1'

ВРЕМЯ АО 5

ВОПРОС 'НАЧАЛО K1 1600 И ВХОД K1 K2 И ЯЧЕЙКА K2 K3'

НАЧАЛО ПО 1600 И ВХОД ПО П7 И ЯЧЕЙКА П7 P(1)

НАЧАЛО ПО 1600 И ВХОД ПО П8 И ЯЧЕЙКА П8 P(2)

НАЧАЛО ПО 1600 И ВХОД ПО П9 И ЯЧЕЙКА П9 P(3)

ВОПРОС 'ВЫХОД ПО K1 И ЯЧЕЙКА K1 K2'

ВЫХОД ПО П10 И ЯЧЕЙКА П10 P(4)

ВОПРОС 'РОД K1 УГОЛ'

РОД Y-ABC УГОЛ

РОД Y-ABE УГОЛ

РОД Y-CBE УГОЛ

РОД Y-BAF УГОЛ

РОД Y-BAG УГОЛ

РОД Y-CBF УГОЛ

РОД Y-ECD УГОЛ

РОД Y-BFE УГОЛ

РОД Y-DEF УГОЛ

РОД Y-EDG УГОЛ

РОД Y-EFG УГОЛ

РОД Y-ABF УГОЛ
РОД Y-BCD УГОЛ
РОД Y-BCE УГОЛ
РОД Y-CDG УГОЛ
РОД Y-CDE УГОЛ
РОД Y-FAG УГОЛ
РОД Y-AGF УГОЛ
РОД Y-DGF УГОЛ
РОД Y-AFB УГОЛ
РОД Y-AFG УГОЛ
РОД Y-BEF УГОЛ
РОД Y-BEC УГОЛ
РОД Y-CED УГОЛ
РОД Y-EBF УГОЛ

ВОПРОС 'K1 ABD K2'

ЭЛМ ABD СТЦ
ВЕРШ ABD A
ВЕРШ ABD D
УГОЛ3 ABD Y-ABE
УГОЛ2 ABD Y-BAG
УГОЛ1 ABD Y-EDG
РОД ABD ТРЕУГОЛЬНИК
СТОРОНА ABD AB
ВЕРШ ABD B
СТОРОНА ABD AD
СТОРОНА ABD BD
ПЛОЩ ABD S-ABD

ВОПРОС 'РАВНО K1 K2 И ЭЛМ K1 СТЦ'

РАВНО Y-BFE Y-AFG И ЭЛМ Y-BFE СТЦ
РАВНО Y-DEF Y-BEC И ЭЛМ Y-DEF СТЦ
РАВНО Y-EFG Y-AFB И ЭЛМ Y-EFG СТЦ
РАВНО AF CF И ЭЛМ AF СТЦ
РАВНО CF AF И ЭЛМ CF СТЦ
РАВНО BE ED И ЭЛМ BE СТЦ
РАВНО ED BE И ЭЛМ ED СТЦ
РАВНО Y-AFB Y-EFG И ЭЛМ Y-AFB СТЦ
РАВНО Y-AFG Y-BFE И ЭЛМ Y-AFG СТЦ
РАВНО Y-BEF Y-CED И ЭЛМ Y-BEF СТЦ
РАВНО Y-BEC Y-DEF И ЭЛМ Y-BEC СТЦ
РАВНО Y-CED Y-BEF И ЭЛМ Y-CED СТЦ

ЗАДАЧА: ДАНО BF И FE И BE И EC, НАЙТИ AD,
ОПТИМИЗИРУЯ РАЗМЕР, НАЧАЛО ПРОГРАММЫ 500

ЗАДАЧА РЕШЕНА, ПРОГРАММА С 500 ДО 562

ЗАПУСК: ПРИ BF = 5.9 И EF = 1.8 И BE = 5.5 И CE = 2.4,
НАЧАЛО ПРОГРАММЫ 500

AD = .786618239E1

9000 ПУСТЬ K = 498

9001 ВЫЧИСЛИ K = K + 2

9002 ЗАШЛИ КОМАНДУ K В A(K) КАК СТРОКУ

9003 ЕСЛИ НЕВОЗМОЖНО НА 9011

9004 ОБРАЗУИ В М ИЗ A(K) ОТ 5 ДО 11

9005 ЕСЛИ М НЕ-РАВНО 'ВЫПОЛНИ' НА 9001

9006 ОБРАЗУИ В М ИЗ A(K) ОТ 13 ДО 16

9007 СОЕДИНИ 'НАЧАЛО K1' И М В М

9008 СОЕДИНИ М И 'И МОДУЛЬ K1 K2' В М

9009 ВОПРОС М

9010 НА 9001

9011 ВОЗВРАТ

ВЫПОЛНИ 9000

НАЧАЛО ДО 1300 И

МОДУЛЬ ДО ПО-ТРЕМ-СТОРОНАМ-ТРЕУГОЛЬНИКА-ВЫЧИСЛЯЕТ-ЕГО-УГЛЫ,

ДЛИНА-ПРОГРАММЫ-16

НАЧАЛО ГО 1500 И

МОДУЛЬ ГО ВЫЧИСЛЯЕТ-УГОЛ, ДОПОЛНЯЮЩИИ-ДАННЫИ-ДО-180-ГРАДУСОВ,

ДЛИНА-ПРОГРАММЫ-2

НАЧАЛО ЕО 1100 И

МОДУЛЬ ЕО ВЫЧИСЛЯЕТ-СУММУ-ДЛИН-ДВУХ-ОТРЕЗКОВ,

ДЛИНА-ПРОГРАММЫ-2

НАЧАЛО НО 1233 И

МОДУЛЬ НО ПРИСВАИВАЕТ-ЗНАЧЕНИЕ-ОДНОЙ-ВЕЛИЧИНЫ-

ДРУГОЙ-ПРИ-РАВЕНСТВЕ-ЭТИХ-ВЕЛИЧИН, ДЛИНА-ПРОГРАММЫ-2

НАЧАЛО ЕО 1100 И

МОДУЛЬ ЕО ВЫЧИСЛЯЕТ-СУММУ-ДЛИН-ДВУХ-ОТРЕЗКОВ,

ДЛИНА-ПРОГРАММЫ-2

НАЧАЛО НО 1233 И

МОДУЛЬ НО ПРИСВАИВАЕТ-ЗНАЧЕНИЕ-ОДНОЙ-ВЕЛИЧИНЫ-

ДРУГОЙ-ПРИ-РАВЕНСТВЕ-ЭТИХ-ВЕЛИЧИН, ДЛИНА-ПРОГРАММЫ-2

НАЧАЛО АО 1000 И

МОДУЛЬ АО ПО-ДВУМ-СТОРОНАМ-ТРЕУГОЛЬНИКА-И-УГЛУ-

МЕЖДУ-НИМИ-ВЫЧИСЛЯЕТ-ОСТАЛЬНЫЕ-УГЛЫ-И-ТРЕТЬЮ-

СТОРОНУ, ДЛИНА-ПРОГРАММЫ-16

ПРЕДПИСАНИЕ 'ЧТЕ СЕМАНТИКА'

ПОНЯТНО

СИТУАЦИЯ: ПРЯМАЯ - A B, ПРЯМАЯ - A C, ПРЯМАЯ - B D C,

ПРЯМАЯ - F D A, РАВНЫ BD AC, ЗНАЧЕНИЕ AD 3.5,

ЗНАЧЕНИЕ DC 2.1

ЗАДАЧА: ДАНО Y-BDF, НАЙТИ BA И S-BDA,

НАЧАЛО ПРОГРАММЫ 700

ЗАДАЧА РЕШЕНА, ПРОГРАММА С 700 ДО 776

ЗАПУСК: ПРИ $Y-BDF = 1.57$, НАЧАЛО ПРОГРАММЫ 700

$AB = .537891000E1$

$S-ABD = .714040100E1$

ЗАДАЧА: ДАНО $Y-BDF$, НАЙТИ FD , НАЧАЛО ПРОГРАММЫ 900

РЕШЕНИЕ ЗАДАЧИ НЕ НАЙДЕНО

ЗАДАЧА: ДАНО $Y-BDF$, НАЙТИ DA И DC , НАЧАЛО ПРОГРАММЫ 900
ИСКОМЫЕ ВЕЛИЧИНЫ ИЗВЕСТНЫ

Комментарий. В приведенном примере диалога работа начинается с того, что из архива операционной системы считываются разделы, содержащие описание семантики модулей, правила вывода, представляющие знания о геометрии, и тексты программ модулей. Затем задается оператор, описывающий ситуацию, чертеж которой изображен на рис.6.4. На этом чертеже ставится задача: составить

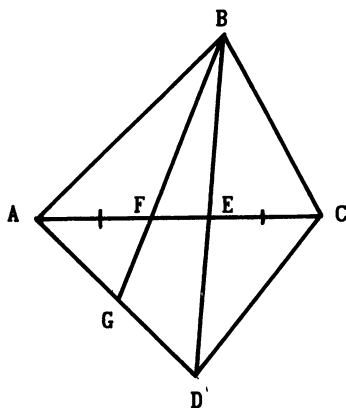


Рис.6.4

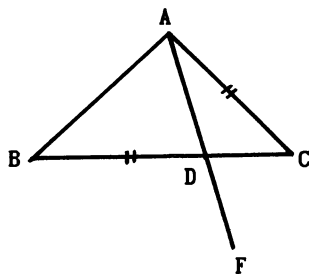


Рис 6 5

программу, вычисляющую площадь треугольника CDE при заданных отрезках AB, AF, FE и угле BAF, учитывая, что в качестве параметра оптимизации задано время. Далее выдается текст полученной программы, которая содержит обращения к модулям, начинающимся с операторов с адресами: 1000, 1500, 1233, 1400 и 1600. Модуль с адресом начала 1000 вычисляет по двум сторонам и углу треугольника два других угла и третью сторону. Модуль с адресом начала 1500 вычисляет угол, дополняющий данный до 180 градусов. Модуль с адресом начала 1233 осуществляет присваивание

значения одной величины другой в случае их равенства. Модуль с адресом начала 1400 вычисляет разность двух отрезков. Модуль с адресом начала 1600 по трем сторонам треугольника вычисляет его площадь. Таким образом, эта программа реализует решение поставленной задачи, в соответствии с которым сначала вычисляются отрезок BF и угол AFB, далее последовательно вычисляются угол BFE, отрезок FC, отрезок EC, отрезок BE и угол BEF, отрезок ED, угол CED, отрезок DC и, наконец, площадь треугольника CDE.

Следующий оператор – это один из вычислительных операторов языка системы «ВОПРОС-ОТВЕТ-2», использование которого демонстрирует возможность проведения промежуточных расчетов при работе с рассматриваемым пакетом. Затем ранее сгенерированная программа дважды запускается на счет с разными начальными значениями.

Последующие операторы используются для получения сведений о модулях, входящих в функциональное наполнение пакета, и о предметной области. В частности, выдается текст программы одного из модулей и запрашивается информация о соответствии имен входов и выходов другого модуля, использованных при описании его семантики, с программными идентификаторами. Далее распечатывается список углов, порожденных в результате выполнения оператора описания ситуации. А затем выдаются ответы на вопросы о свойствах некоторого треугольника и о том, какие элементы ситуации равны между собой по величине.

Следующие два оператора задают постановку новой задачи в рамках той же самой ситуации и запуск полученной программы на счет.

Далее задается и выполняется программа на языке системы «ВОПРОС-ОТВЕТ-2», состоящая из 12 операторов, которая представляет сложный запрос к базе данных. В результате выполнения этого запроса из базы данных извлекаются и распечатываются тексты комментариев, описывающих модули, вызываемые в сгенерированной программе с адресом начала 500. Эти комментарии выдаются в том порядке, в котором выполняются описываемые ими модули, и, таким образом, поясняется найденный системой способ решения поставленной задачи.

Затем база данных обновляется, т.е. из нее выбрасываются сведения о чертеже, представленном на рис.6.4, и выполняется оператор, задающий описание новой простой ситуации, чертеж которой представлен на рис.6.5.

В новой ситуации ставится и просчитывается очередная задача. При попытке решить следующую задачу обнаружива-

ется, что она не имеет решения. И, наконец, при выполнении последнего в данном примере оператора устанавливается, что искомые величины уже определены при описании ситуации.

6.3.6. Связь с экспертными системами. В заключение, необходимо отметить, что методический пакет МП явился одной из первых по времени разработок, получивших позднее название гибридных экспертных систем. От классических экспертных систем гибридные отличаются тем, что используют по крайней мере два принципиально разных способа представления знаний. Как правило, это продукции или фреймы и процедуры. Именно такой подход используется в пакете МП, где знания о предметной области описываются двумя способами: в виде собственно расчетных модулей (процедур), а также их семантики, заданной фактами и правилами вывода (продукциями). Подобный подход позволяет использовать расчетные и моделирующие программы в экспертных системах или, иначе, применять декларативные знания в пакетах прикладных программ. Это дает возможность существенно повысить уровень компетентности и расширить сферу применения таких гибридных программных систем.

В качестве примера сошлемся на проект HAL-1986 [140]. Эта гибридная система предназначена для диагностики аварий на АЭС и анализирует показания примерно 1000 датчиков. Экспертная часть системы использует продукционную модель и реализована на языке Лисп на ЭВМ PDP-11. Экспертная часть будет соединена с системой численного моделирования, причем расчеты на модели будут проходить в десять раз быстрее реальных процессов на АЭС. Для достижения такой производительности для обработки процедурных знаний будут применены четыре процессора супер-ЭВМ Cray X-MP/24, каждый производительностью 500 миллионов операций с плавающей запятой в секунду.

Тенденция разделения аппаратных средств поддержки знаний разного типа, как в проекте HAL-1986, находит свое развитие и в создании экспериментальных ПЭВМ с двумя процессорами, классическим «фон-неймановским» для процедурных знаний и специализированным (например, Лисп-или Пролог-машиной) для обработки продукций или фреймов. Наличие таких аппаратных средств позволяет надеяться на дальнейшее развитие гибридных систем, подобных пакету МП.

ЗАКЛЮЧЕНИЕ

Авторы далеки от мысли о том, что рассмотренными пакетами исчерпывается все многообразие языковых и системных решений в области прикладных программных разработок. Несомненно, однако, что создание новых прикладных систем следует начинать со знакомства с существующими аналогами, избегая тем самым непроизводительных затрат, связанных с «изобретением велосипеда». Помочь такому знакомству и была призвана настоящая книга.

Для продолжения знакомства с отечественными разработками по данной тематике можно рекомендовать ряд сборников «Пакеты прикладных программ», вышедших в издательстве «Наука» в серии «Алгоритмы и алгоритмические языки» (главный редактор – академик А.А.Самарский) и в Сибирском отделении издательства «Наука» (редакторы – академик В.М.Матросов и О.Г.Диваков).

СЛОВАРЬ ТЕРМИНОВ

Активная дисциплина – способ формирования расчетной программы, предусматривающий модификацию существующих или же создание новых модулей функционального наполнения пакета.

Базовый язык – язык программирования, на котором пишутся модули функционального наполнения.

Безболезненное развитие – организация разработки функционального наполнения, позволяющая избежать изменений существующих программ при подключении к пакету новых модулей.

Вычислительный эксперимент – метод изучения устройств или физических процессов посредством построения их математической модели и последующего численного исследования этой модели, позволяющего «пронграть» их поведение в различных условиях.

Дисциплина работы – система правил, соглашений, технологических подходов и приемов, принятых при разработке, отладке и эксплуатации программ.

Инструментальный язык – язык, на котором пишутся программы системного наполнения.

Информационное обслуживание – удовлетворение информационных запросов, поступающих от пользователей или от программных компонентов пакета.

Каркасное конструирование – способ формирования расчетных программ, при котором все многообразие их конфигураций достигается за счет заполнения гнезд фиксированного каркаса различными модулями функционального наполнения.

Конечный пользователь – эксплуатирующий пакет специалист в предметной области, не обязательно имеющий высокий уровень подготовки в области вычислительной техники и программирования.

Межмодульный интерфейс – организация информационных и управляющих связей между модулями.

Модуль – конструктивный элемент, который может использоваться на различных стадиях функционирования пакета.

Модуляризация – разбиение функционального наполнения пакета на модули.

Пакет прикладных программ (пакет) – комплекс взаимосвязанных прикладных программ и средств системного обеспечения (программных и языковых), предназначенный для автоматизации решения определенного класса задач.

Пассивная дисциплина – способ формирования расчетной программы, исключающий модификацию пользователем модулей функционального наполнения пакета.

Планирование вычислений – определение состава и порядка выполнения входящих в расчетную программу модулей функционального наполнения при цепочечном конструировании программ.

Покрытие (охват) предметной области – создание набора модулей функционального наполнения, позволяющего для любой задачи из данной области построить решающую ее программу, целиком сконструированную из некоторого подмножества этого набора модулей.

Предметная область – совокупность решаемых прикладных задач и используемых при этом численных методов.

Прикладная деятельность – круг работ, связанных с созданием алгоритмов и программ решения задач, а также с подготовкой и проведением расчетов.

Регламент модуляризации – методы разбиения программного материала на модули, а также правила оформления модулей.

Системное наполнение – совокупность программ, поддерживающих принятую в данной прикладной деятельности дисциплину работы и обеспечивающих тем самым взаимодействие пользователя с пакетом.

Функциональное наполнение – совокупность модулей пакета, отражающих специфику обслуживаемой им предметной области и используемых как компоненты формируемых расчетных программ.

Цепочечное конструирование – формирование расчетной программы в виде цепочки выполняемых друг за другом модулей функционального наполнения.

Язык заданий – средство общения пользователя с пакетом, позволяющее формировать запросы на различные виды работ, выполняемых в рамках прикладной деятельности.

Язык запросов – ориентированный на конечного пользователя язык заданий пакета, позволяющий формировать запросы, указывающие, «что необходимо получить» без явного указания того, «как это получить».

Язык сборки – язык заданий пакета, обеспечивающий непосредственное управление процессами конструирования и исполнения расчетной программы, а также, возможно, предоставляющий средства развития или модификации функционального наполнения.

СПИСОК ЛИТЕРАТУРЫ

1. Самарский А. А. Вычислительный эксперимент и требования к программным средствам // Промышленная технология создания и применения программных средств в организационном управлении и НИОКР: Материалы Всесоюзного семинара.- Свердловск, 1984.
2. Самарский А. А. Пакеты прикладных программ как средство обеспечения сложных физических расчетов // Перспективы системного и теоретического программирования: Труды Всесоюзного симпозиума.- Новосибирск: ВЦ СО АН СССР, 1978.
3. Карпов В. Я., Корягин Д. А., Самарский А. А. Принципы разработки пакетов прикладных программ для задач математической физики // Журн. вычисл. математики и мат. физики.- 1978.- Т.18, № 2.- С.458-467.
4. Ершов А. П., Ильин В. П. Пакеты программ - технология решения прикладных задач.- Новосибирск: ВЦ СО АН СССР, 1978.
5. Загацкий Б. А. Прикладные программы для ЕС ЭВМ СССР // Системное программирование. Т.2.- Новосибирск: ВЦ СО АН СССР, 1973.
6. Тамм Б. Г., Тыугу Э. Х. Пакеты программ // Изв. АН СССР. Техн. кибернетика.- 1977.- № 5.- С.111-124.
7. Сергиенко И. В. Вопросы построения и использования математического обеспечения методов оптимизации.- Препр. / ИК АН УССР.- Киев, 1979.- № 79-38.
8. Фатеев А. Е., Ройтман А. И., Фатеева Т. П. Прикладные программы в системе математического обеспечения ЕС ЭВМ.- М.: Статистика, 1978.
9. Яиенко Н. Н., Карначук В. И., Коновалов А. Н. Проблемы математической технологии // Численные методы механики сплошной среды: Сб.

- статей. Т.8, №3 / Под ред. Н.Н.Яненко.- Новосибирск, 1977.- С.129-157.
10. Пакеты прикладных программ. Методы и разработки. / Под ред. В.М.Матросова.- Новосибирск: Наука, 1981.
 11. Программные системы. / Под ред. П.Бахманна. / Пер. с немец. под ред. И.В.Поттосина.- М.: Мир, 1988.- 288 с.
 12. Академия наук СССР. Комиссия по пакетам прикладных программ координационного комитета по вычислительной технике. Положение о порядке разработки и документировании пакетов прикладных программ. Проект.- М.: ВЦ АН СССР, 1981.
 13. Карпов В. Я., Корягин Д. А. Задачи системного программирования, связанные с разработкой пакетов прикладных программ // Перспективы системного и теоретического программирования: Труды Всесоюзного симпозиума.- Новосибирск: ВЦ СО АН СССР, 1978.
 14. Самарский А. А. Математическое моделирование и вычислительный эксперимент // Вестник АН СССР.- 1979.- № 5.- С.38-49.
 15. Комплексы программ математической физики и архитектура ЭВМ. Труды школы-семинара, п.Шушинское. / Под ред. Ю.И.Шокина.- Красноярск, 1988.
 16. Тихонов А. Н., Арсенин В. Я. Методы решения некорректных задач.- М.: Наука, 1986.- 287 с.
 17. Тихонов А. Н., Арсенин В. Я., Думова А. А. и др. О многоцелевой проблемно-ориентированной системе обработки результатов экспериментов.- Препр. / ИПМ АН СССР.- М., 1976.- № 142.
 18. Говорун Н. Н., Дорж Л., Иванов В. Г., Лукьянцев А. Ф. Модульная система программ обработки экспериментальных данных // Программирование и математические методы решения физических задач.- Дубна: ОИЯИ, 1974.
 19. Говорун Н. Н., Иванченко И. М., Нефедьева Л. С.. Диалог в системах автоматизированной обработки данных // Управляющие системы и машины.- 1974.- № 1.- С.8-13.
 20. Говорун Н. Н., Иванченко И. М. Математическое обеспечение многоабонентной системы для обмера снимков с трековых камер // Программирование.- 1976.- № 4.- С.52-65.

21. Бутцева Г. Л., Кениг Х., Нефедьева Л. С. и др. Система обработки спектров на ЭВМ БЭСМ-6 // Программирование и математические методы решения физических задач.- Дубна: ОИЯИ, 1977.
22. Тамм Б. Г., Тыугу Э. Х. О создании проблемно-ориентированного программного обеспечения // Кибернетика.- 1975.- № 4.- С.76-85.
23. Сб. научных программ на Фортране. Статистика. Вып.1.- М.: Статистика, 1974.
24. Сб. научных программ на Фортране. Матричная алгебра и линейная алгебра. Вып.2.- М.: Статистика, 1974.
25. Сб. «Стандартные программы для машин типа М-20». № 1.- М.: ВЦ АН СССР, 1965.
26. Галактионов В. В., Лукстиня Л. А., Панченко Л. М. и др. Библиотека программ на Фортране. Подробное описание. Т.1.- Дубна: ОИЯИ, 1970.
27. Арушанян О. Б. Автоматизация конструирования библиотек программ.- М.: Изд-во МГУ, 1988.- 248 с.
28. Гуляев Ю. П., Петров В. И. Формирование библиотеки программных модулей для множества проблемно-ориентированных задач // Программирование.- 1976.- № 6.- С.61-67.
29. Арушанян О. Б. Центральные проблемы конструирования библиотек программ общего назначения // Библиотеки программ. Архитектура и наполнение.- М.: Изд-во МГУ, 1981.
30. Коновалов А. Н., Яненко Н. Н. Модульный принцип построения программ как основа создания пакета прикладных программ решения задач механики сплошной среды // Комплексы программ математической физики.- Новосибирск, 1972.
31. Ross D. T. Reflections on requirements // IEEE Transactions on Software Engineering.- 1977.- V.SE-3, № 1.
32. Коновалов А. Н. Модульный анализ вычислительного алгоритма в задаче планового вытеснения нефти водой // Труды III семинара по комплексам программ математической физики.- Новосибирск: ВЦ и Ин-т теорет. и прикл. мех. СО АН СССР, 1973.- С.81-94.

33. Азаров С. С. Модульный анализ задач управления запасами // Вычислительные аспекты в пакетах прикладных программ.- Киев: ИК АН УССР, 1980.
34. Symposium of modular coding for reactor calculation. Newsletter of the ENEA Computer Programme Library, 1971, № 11, March.
35. THE CPC PROGRAM LIBRARY // Computer physics communication.- 1970.- № 1.
36. Los Alamos library // Los Alamos Scientific Laboratories, 1971.
37. A catalog of Control Data software. Control Data Corporation.- Minneapolis; Minnesota, 1974.
38. Жуков В. В., Азеев А. А., Егельский А. И. Организация работ в фондах алгоритмов и программ.- М.: Статистика, 1980.- 72 с.
39. Родионов С. Т. Основные концепции модульности и анализ некоторых направлений развития общей технологии программирования // Программирование.- 1980.- № 2.- С.31-37.
40. Каминский Л. Г., Клименко С. В., Лукьянцев А. Ф. и др. PATCHY - система хранения, модернизации и эксплуатации больших программ // Труды IV Всесоюзного семинара по комплексам программ математической физики.- Новосибирск, 1976.
41. Бежанова М. М. Входные языки пакетов прикладных программ.- Препр. / ВЦ СО АН СССР.- Новосибирск, 1979.- № 168.
42. Бежанова М. М. Встроенные проблемно-ориентированные системы.- Препр. / ВЦ СО АН СССР.- Новосибирск, 1979.- № 205.
43. Ильин В. П., Катешов В. А. Автоматизация описания двумерных краевых задач.- Препр. / ВЦ СО АН СССР.- Новосибирск, 1979.- № 173.
44. Михалевич В. С., Сергиенко И. В., Быков В. Н. и др. О математическом обеспечении пакета программ ДИСПРО для решения задач дискретного программирования.- Препр. / ИК АН УССР.- Киев, 1979.- № 79-3.
45. Глушков В. М. Об одном методе автоматизации программирования // Проблемы кибернетики: Сб. статей. Вып.2 / Под ред. А.А.Ляпунова.- М.: Наука, 1959.- С.181-184.

46. Кахро М. И., Калья А. П., Тыгу Э. Х. Инструментальная система программирования ЕС ЭВМ (ПРИЗ).— М.: Финансы и статистика, 1988.— 180 с.
47. Бабаев И. О., Новиков Ф. А., Петрушина Т. И. Язык Декарт — входной язык системы СПОРА // Прикладная информатика: Сб. статей. Вып. 1. / Под ред. В. М. Савинкова.— М.: Финансы и статистика, 1981.— С. 35–73.
48. Тыгу Э. Х. Концептуальное программирование.— М.: Наука, 1984.— 255 с.
49. Мельников И. А., Мартин К. О., Пруден Э. В. и др. Метасистема для создания информационно-связанных специализированных систем программирования // Кибернетика.— 1974.— № 6.— С. 69–73.
50. Montgomery G. R. PLAN — the engineer defines his own programming language. // Intern. Conf. Comput. Aided Design, Southampton, 1969. — London: IEE, 1969.
51. Шура-Буря М. Р. Интерпретирующая система ИС-2 для М-20 // Стандартные программы для машин типа М-20, № 1.— М.: ВЦ АН СССР, 1965.— С. 5–24.
52. Жоголев Е. А. Система программирования с использованием библиотеки подпрограмм. // Система автоматизации программирования.— М.: Физматгиз, 1961.
53. Мартынюк В. В. О методе символических адресов // Проблемы кибернетики: Сб. статей. Вып. 6 / Под ред. А. А. Ляпунова.— М.: Наука, 1961.— С. 45–58.
54. Фролов А. С., Ченцов Н. Н. Некоторые вопросы конструирования случайных экспериментов // Методы программирования и решения задач на ЦВМ, 1959.— С. 140–147.
55. Молчанов И. Н., Николаенко Л. Д., Кириченко М. П. Об одном пакете программ для решения систем линейных алгебраических уравнений // Кибернетика.— 1972.— № 1.— С. 127–133.
56. Воеводин В. В., Гайсарян С. С., Кабанов М. И. Автоматизированная генерация программ // Численный анализ на Фортране: Сб. статей. Вып. 1.— М.: Изд-во МГУ, 1973.— С. 3–13.
57. Жоголев Е. А. Принципы построения многоязыковой системы модульного программирования // Кибернетика.— 1974.— № 4.— С. 1–5.

58. Зизин М. Н., Загацкий Б. А., Темноева Т. А., Ярославцева Л. Н. Автоматизация реакторных расчетов.- М.: Атомиздат, 1974.
59. Столяров Л. Н. Краткий обзор принципов организации пакетов программ // Труды IV Всесоюзного семинара по комплексам программ математической физики.- Новосибирск, 1976.
60. Стукало А. С. Некоторые требования к пакетам прикладных программ. (Обзор и анализ) // Труды IV Всесоюзного семинара по комплексам программ математической физики.- Новосибирск, 1976.
61. Лавров С. С. Использование вычислительной техники, программирование и искусственный интеллект // Промышленная технология создания и применения программных средств в организационном управлении и НИОКР: Материалы Всесоюзного семинара.- Свердловск, 1984.
62. Ильин В. П. Об одном варианте системы модульного программирования.- Препр. / ВЦ СО АН СССР.- Новосибирск, 1976.- № 9.
63. Roberts K. V. An introduction to OLYMPUS system // Computer physics communication.- 1974.- V.7.- P.237-243.
64. Christiansen J. P., Roberts K. V. OLYMPUS. A standart and utility package for initial-value Fortran programs // Computer physics communication.- 1974.- V.7.- P.245-270.
65. Горбунов-Посадов М. М., Карпов В. Я., Корягин Д. А. и др. Пакет прикладных программ САФРА. Системное наполнение.- Препр. / ИПМ АН СССР.- М., 1977.- № 85.
66. Светлакова Ф. Г. Специализированный язык взаимодействия с информационной базой пакета программ.- Препр. / ВЦ СО АН СССР.- Новосибирск, 1980.- № 254.
67. Вольдман Г. Ш., Задыхайло И. Б. Некоторые соображения об определении степени неprocedureности языков программирования.- Препр. / ИПМ АН СССР.- М., 1977.- № 51.
68. Дейкстра Э. Дисциплина программирования. / Пер. с англ. под ред. Э.З.Любимского.- М.: Мир, 1978.- 274 с.

69. Parnas D. L. On the criteria to be used in decomposing systems into modules // Comm. ACM.- 1972.- V.15, № 12.- P.1053-1058.
70. Попов Э. В. Экспертные системы: Решение неформализованных задач в диалоге с ЭВМ.- М.: Наука, 1987.- 284 с.
71. Бухштаб Ю. А., Горлин А. И., Камынин С. С. и др. Интеллектуальный пакет, использующий при планировании вычислений знания о предметной области и функциональных модулях // Изв. АН СССР. Техн. кибернетика.- 1981.- № 5.- С.113-124.
72. Брукс Ф. П. мл. Как проектируются и создаются программные комплексы: Мифический человеко-месяц. Очерки по системному программированию / Пер с англ. под ред. А.П.Ершова.- М.: Наука, 1979.- 151 с.
73. Зуев В. И., Крюков В. М., Легоньков В. И. Управление данными в вычислительном эксперименте.- М.: Наука, 1986.- 157 с.
74. Воронков А. В., Зименков В. И., Легоньков В. И. и др. Система обеспечения комплексов программ математической физики.- Препр. / ИПМ АН СССР.- М., 1979.- № 49.
75. Басс Л. П., Гермогенова Т. А., Журавлев В. И. и др. Система ПНФ. Принципы создания пакетов программ на базе нестандартизированного программного фонда.- Препр. / ИПМ АН СССР.- М., 1980.- № 154.
76. Борисов В. М. Принципы реализации компонентов системного обеспечения ВА-языка // Библиотека программ. Архитектура и наполнение.- М.: Изд-во МГУ, 1981.
77. Kempf J. Numerical software tools in C. - Englewood Cliffs (N.J.): Prentice-Hall, 1987. - 261 p.
78. Cardenas A. F., Karplus W. J. PDEL - a language for partial differential equations // Comm. ACM.- 1970.- V.13, № 3.
79. Ross D. T. The AED Approach to Generalized Computer - Aided Design // Proceedings of ACM 22-nd National Conference, 1977.
80. Тамм Б. Г. Описание языка САП-2 для программирования работы станков с ЧПУ // Алгоритмы и алгоритмические языки: Сб. статей. Вып.5.- М.: ВЦ АН СССР, 1971.- С.67-78.

81. Прууден Ю. И. АПРОКС – специализированный язык с проблемной ориентацией // Алгоритмы и алгоритмические языки: Сб. статей. Вып.5.– М.: ВЦ АН СССР, 1971.– С.79–93.
82. Парасюк И. Н., Сергиенко И. В. Пакет программ анализа данных: технология разработки.– М.: Финансы и статистика, 1988.
83. Сергиенко И. В., Парасюк И. Н., Тукалевская Н. И. Автоматизированные системы обработки данных.– Киев: Наукова думка, 1976.– 256 с.
84. Бежанова М. М. Системная программа ТЕНЗОР // Труды второй Всесоюзной конференции по программированию, заседание Н.– Новосибирск, 1970.
85. Dokumentation zum Programmsystem MAIWEP. Karl-Marx-Stadt, Technische Hochschule Karl-Marx-Stadt, 1978.
86. Тыугу Э. Х. Решение задач на вычислительных моделях // Журн. вычисл. математики и мат. физики.– 1970.– Т.10, № 3.– С.716–733.
87. Тыугу Э. Х. Решатель вычислительных задач // Журн. вычисл. математики и мат. физики.– 1971.– Т.11, № 4.– С.992–1004.
88. Башмаков И.А., Бесфамильный М. С. Математический аппарат описания модели предметной области пакетов прикладных программ // Труды МЭИ. Вып.221.– М., 1975.
89. Белкин А. Р., Медведев А. Е. Планирование вычислений в системах автоматизированного проектирования // Труды XXII научной конференции Моск. физ.-техн. ин-та 1975 г. Серия «Аэрофиз. и прикл. мат.» – Долгопрудный, 1976.
90. Брябрин В. М. Диалоговая информационно-логическая система // Модели данных и системы баз данных.– М.: Наука, 1979.
91. Бухштаб Ю. А. Система «ВОПРОС-ОТВЕТ» (краткое описание и входной язык).– Препр. / ИПМ АН СССР.– М., 1975.– № 104.
92. Бухштаб Ю. А., Камынин С. С. Система «ВОПРОС-ОТВЕТ-2» и ее возможности.– Препр. / ИПМ АН СССР.– М., 1978.– № 50.
93. Нильсон Н. Искусственный интеллект / Пер. с англ. под ред. С.В.Фомина.– М.: Мир, 1973.– 270 с.
94. Бухштаб Ю. А., Горлин А. И., Камынин С. С. и др. Об одном методе планирования рас-

- четных цепочек // Программирование.- 1981.- № 3.- С.34-38.
95. Корухова Л. С., Любимский Э. З. Система планирования действий, основанная на некоторых естественных принципах - ПЛАР.- Препр. / ИПМ АН СССР.- М., 1978.- № 53.
96. Литвинцев Н. И. Планирование вычислений в диалоговой системе долгосрочного планирования // Изв. АН СССР. Техн. кибернетика.- 1980.- № 4.- С.21-28.
97. Горбунов-Посадов М. М., Хиздер Л. А. Система ОХРА. Основные понятия. Базовые операции.- Препр. / ИПМ АН СССР.- М., 1979.- № 67.
98. Загацкий Б. А., Потаманов В. И., Темноева Т. А. Архив. Система хранения информации в памяти ЭВМ.- Препр. / НИИАР.- Мелекес, 1969.- № П-63.
99. Advanced programming environments. Proceedings of an international workshop. Edited by Goos G. and Hantmanis J. // Lecture notes in computer science. Vol.244. Berlin: Springer-verlag, 1986. - 604 p.
100. Жук В. И., Малашин И. И. Системы баз данных, ориентированные на применение при автоматизации физических экспериментов. Состояние проблемы. Обзор.- М.: ЦНИИАтоминформ, 1986.- 61 с.
101. Королев Л. Н. Структура ЭВМ и их математическое обеспечение.- М.: Наука, 1978.- 351 с.
102. Баула В. Г. Информатор: автоматизированная документально-справочная система // Вопросы конструирования библиотек программ.- М.: Изд-во МГУ, 1980.
103. Горбунов-Посадов М. М., Карпов В. Я., Корягин Д. А., Красотченко В. В. Информационное обеспечение пакета САФРА.- Препр. / ИПМ АН СССР.- М., 1981.- № 172.
104. Сакман Г. Решение задач в системе человек - ЭВМ. / Пер. с англ. под ред. О.К.Тихомирова.- М.: Мир, 1973.- 352 с.
105. Довгялло А. М. Диалог пользователя и ЭВМ. Основы проектирования и реализации.- Киев: Наукова думка, 1981.- 232 с.
106. Мальковский М. Г. Диалог с системой искусственного интеллекта.- М.: Изд-во МГУ, 1985.

107. Перевозчикова О. Л., Ющенко Е. Л. Системы диалогового решения задач на ЭВМ. — Киев: Наукова думка, 1986. — 264 с.
108. Алфёрова М. И., Горбунов-Посадов М. М., Матекин М. П., Нефедова Р. А. Средства документирования в пакете Сафра. — Препр. / ИПМ АН СССР. — М., 1981. — № 27. — 13 с.
109. Sherrington H. What is an application package? // Eur. Comput. Congr. Conf. Proc., Brunel University, 1974, Brunel.
110. Воронков А. В., Горлин А. И., Загачкий Б. А., Легоньков В. И. Математическое обеспечение ЭВМ. Проект представления модуля прикладной программы. — Новосибирск: ВЦ СО АН СССР, 1975.
111. Воронков А. В., Карпов В. Я., Шарев Д. Я. Оформление документации программ для решения задач математической физики. — М.: ИПМ АН СССР, 1977.
112. Воронков А. В., Фридрих Ф. Вопросы подготовки информационных материалов для программ, библиотек и пакетов программ задач математической физики. — М.: ВЦ АН СССР, 1979.
113. Пайл Л. Ввод данных путем вопросов и ответов // Современное программирование. Мультипрограммирование и разделение времени. / Пер. с англ. под ред. И. Б. Задыхайло, Э. З. Любимского, В. В. Мартынюка. — М.: Мир, 1970. — С. 315–325.
114. Уинстон П. Искусственный интеллект. / Пер. с англ. под ред. Д. А. Поспелова. — М.: Мир, 1980. — 519 с.
115. Горбунов-Посадов М. М., Карпов В. Я., Корягин Д. А. и др. Пакет Сафра: программное обеспечение вычислительного эксперимента // Алгоритмы и алгоритмические языки. Пакеты прикладных программ. Вычислительный эксперимент. — М.: Наука, 1983. — С. 12–50.
116. Горбунов-Посадов М. М. Изменяемые программы и однородные модули // Программирование. — 1988. — № 4. — С. 38–49.
117. Горбунов-Посадов М. М. Некоторые межмодульные связи, нуждающиеся в инструментальной поддержке // Автоматизированное рабочее место прог-

раммиста.- Новосибирск: ВЦ СО АН СССР, 1988.- С.86-92.

118. Басс Л. П., Гермогенова Т. А., Журавлев В. И. и др. Система ПНФ. Принципы создания пакетов программ на базе нестандартизированного программного фонда.- Препр. / ИПМ АН СССР.- М., 1980.- № 154.
119. Волощенко А. М., Корягин Д. А., Кротов В. И. и др. Основные этапы разработки программ с помощью системы ПНФ.- Препр. / ИПМ АН СССР.- М., 1981.- № 181.
120. Корягин Д. А., Кротов В. И., Сигалев В. А. Система ПНФ. Технология разработки пакетов прикладных программ.- Препр. / ИПМ АН СССР.- М., 1986.- № 151.- 26 с.
121. Горелик А. М., Корягин Д. А., Кротов В. И., Луховицкая Э. С. Пакет ЗАЩИТА. Общее описание языка заданий.- Препр. / ИПМ АН СССР.- М., 1982.- № 207.- 27 с.
122. Волощенко А. М., Корягин Д. А., Кротов В. И., Луховицкая Э. С. Пакет ЗАЩИТА. Системное и функциональное наполнение.- Препр. / ИПМ АН СССР.- М., 1984.- № 16.- 28 с.
123. Басс Л. П., Волощенко А. М., Кротов В. И., Лукичева Е. В. Пакет ЗАЩИТА. Версия 3.0. Основные возможности функционального и системного наполнения.- Препр. / ИПМ АН СССР.- М., 1987.- № 27.- 24 с.
124. Тихонов А. Н., Арсенин В. Я., Галкин А. Л. и др. Разработка многоцелевых проблемно-ориентированных систем с помощью анализатора PLAN-БЭСМ-6. Система КОРПУСКУЛА.- Препр. / ИПМ АН СССР.- М., 1982.- № 97.
125. Тихонов А. Н., Арсенин В. Я., Коробочкин А. Е. и др. Вычислительный эксперимент при решении обратных задач диагностики плазмы // Алгоритмы и алгоритмические языки. Пакеты прикладных программ. Вычислительный эксперимент.- М.: Наука, 1983.- С.3-11.
126. Митрофанов В. Б. Система автоматизированной обработки на ЭВМ рентгеновских изображений высокотемпературной плазмы (система ЭОС-2).- Препр. / ИПМ АН СССР.- М., 1985.- № 187.

127. Горелышева И. В., Зусман И. Х., Кац Э. Х. и др. Проект реализации универсального анализатора проблемных языков PLAN-БЭСМ-6. - Препр. / ИПМ АН СССР. - М., 1980. - № 47.
128. Горелышева И. В., Кац Э. Х., Коваленко В. Н. и др. Реализация универсального анализатора PLAN-БЭСМ-6. - Препр. / ИПМ АН СССР. - М., 1982. - № 7.
129. Галкин А. Л., Кац Э. Х., Коваленко В. Н. и др. Информационное обеспечение системы КОРПУСКУЛА. - Препр. / ИПМ АН СССР. - М., 1982. - № 125.
130. Горбунов - Посадов М. М., Ермаков А. В., Карпов В. Я. и др. КРИТ - пакет прикладных программ для решения многокритериальных задач оптимального проектирования объектов машиностроения. - Препр. / ИПМ АН СССР. - М., 1985. - № 175. - 29 с.
131. Соболев И. М., Статников Р. Б. Выбор оптимальных параметров в задачах со многими критериями. - М.: Наука, 1981. - 110 с.
132. Волощенко А. М., Гермогенова Т. А., Исаенко Т. Г. и др. Объединенная система константного обеспечения - ОКС. Версия 3.0. - Препр. / ИПМ АН СССР. - М., 1984. - № 20. - 30 с.
133. Владимирова Т. М., Волощенко А. М., Исаенко Т. Г. и др. TERMAS в системе ОКС. - Препр. / ИПМ АН СССР. - М., 1987. - № 193. - 20 с.
134. Базазянц Н. О., Вырский М. Ю., Гермогенова Т. А. и др. АРАМАКО-2F - система обеспечения нейтронными константами расчетов переноса излучения в реакторах и защите. - М.: ИПМ АН СССР, 1976.
135. Абагян А. А., Барыба М. А., Басс Л. П. и др. АРАМАКО-G - система обеспечения многогрупповыми константами расчетов полей гамма-излучения в реакторах и защите. - Препр. / ИПМ АН СССР. - М., 1978. - № 122.
136. Вырский М. Ю., Дубинин А. А., Илюшкин А. И. и др. Многогрупповая система констант для расчета переноса высокоэнергетических нейтронов // Атомная энергия. - 1982. - Т. 53, вып. 2. - С. 113-114.

137. Гомин Е. А., Майоров Л. В. Программа TERMAC // Вопросы атомной науки и техники. Серия «Физика и техника атомных реакторов». – 1982, вып.5.
138. Волощенко А. М., Костин Е. И., Панфилова Е. И., Уткин В. А. РОЗ-6 – система программ для решения уравнения переноса в одномерных геометриях. Версия 2. Инструкция. – М.: ИПМ АН СССР, 1980.
139. Бухштаб Ю. А., Горлин А. И., Камынин С. С. и др. Об одном методе планирования расчетных цепочек // Программирование. – 1981. – № 3. – С.34–38.
140. Лаймен Дж. Новые средства предотвращения аварий на АЭС // Электроника. – 1986. – № 11. – С.3–5.

Научное издание

*ГОРБУНОВ—ПОСАДОВ Михаил Михайлович,
КОРЯГИН Дмитрий Александрович,
МАРТЫНЮК Виктор Владимирович*

СИСТЕМНОЕ ОБЕСПЕЧЕНИЕ
ПАКЕТОВ ПРИКЛАДНЫХ ПРОГРАММ

Серия "Библиотечка программиста", вып. 62

Заведующий редакцией А. С. Косов
Редактор О. И. Сухова
Художественный редактор Т. Н. Кольченко
Технический редактор А. П. Колесникова
Корректор И. Я. Кристаль

ИБ № 41066

Компьютерный набор. Подписано в печать с оригинал-макета 31. 10. 89.
Формат 84х108/32. Бумага **ТИП 2** Гарнитура литературная.
Печать офсетная. Усл. печ. л. 10,92. Усл. кр.-отт. 11,13.
Уч.-изд. л. 12.09. Тираж 27300 экз. Заказ № 191 Цена 75 коп.

Издательско-производственное и книготорговое объединение "Наука"
Главная редакция физико-математической литературы
117071 Москва, В-71, Ленинский проспект, 15

Отпечатано в 4-й типографии ИПКО "Наука"
630077 г. Новосибирск, 77, ул. Станиславского, 25

75 коп.



5B566
Г-676