МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ
БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ
ВЫСШЕГО ПРОФЕССИОНАЛЬНОГО ОБРАЗОВАНИЯ
«САМАРСКИЙ ГОСУДАРСТВЕННЫЙ АЭРОКОСМИЧЕСКИЙ
УНИВЕРСИТЕТ ИМЕНИ АКАДЕМИКА С.П. КОРОЛЕВА
(НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ)»

А.Н. КОВАРЦЕВ, В.В. ЖИДЧЕНКО

МЕТОДЫ И СРЕДСТВА ВИЗУАЛЬНОГО ПАРАЛЛЕЛЬНОГО ПРОГРАММИРОВАНИЯ. АВТОМАТИЗАЦИЯ ПРОГРАММИРОВАНИЯ

Рекомендовано редакционно-издательским советом федерального государственного бюджетного образовательного учреждения высшего профессионального образования «Самарский государственный аэрокосмический университет имени академика С.П. Королева (национальный исследовательский университет)» в качестве учебника для студентов, обучающихся по образовательной программе высшего профессионального образования по направлению подготовки магистров «Фундаментальная информатика и информационные технологии»

САМАРА
Издательство СГАУ
2011

УДК 004.43 (075)
ББК 22.18я7
К 565

Рецензенты: д-р физ.-мат. наук, проф. А.Н. Степанов,
д-р техн. наук, проф. С.А. Прохоров

Коварцев А.Н.

К565 **Методы и средства визуального параллельного программирования. Автоматизация программирования:** учеб. / **А.Н. Коварцев, В.В. Жидченко –** Самара: Изд-во Самар. гос. аэрокосм. ун-та, 2011. – 168 с.

ISBN 978-5-7883-0909-5

Изложены основные сведения, необходимые для работы в области параллельного программирования. Даётся краткая характеристика существующих методов визуального программирования. Рассматриваются основные принципы построения средства автоматизации разработки моделей параллельных алгоритмов на основе визуального стиля программирования, который повышает наглядность представления моделей параллельных алгоритмов, существенно уменьшает число ошибок, допускаемых на этапах проектирования и кодирования программ.

Предназначен для студентов, аспирантов и специалистов, изучающих и практически использующих параллельные компьютерные системы для решения трудоёмких задач.

УДК 004.43 (075)
ББК 22.18я7

ISBN 978-5-7883-0909-5

ОГЛАВЛЕНИЕ

ПРЕДИСЛОВИЕ

Эта книга написана по материалам научных исследований, проводимых авторами в течение нескольких лет в Самарском государственном аэрокосмическом университете имени С.П. Королева (СГАУ), первоначально в области технологий автоматизации программирования, а впоследствии – в области визуального моделирования параллельных вычислений. На кафедре программных систем (СГАУ) это научное направление представлено технологией графосимволического программирования (ГСП), в которой алгоритм программы представляется графом управления, где в вершинах расположены вычислительные модули, а дугам приписаны логические функции.

Основная цель данного учебника заключается в ознакомлении читателей с методами автоматизации программирования и визуального моделирования параллельных вычислительных процессов, которые обладают достоинством наглядного представления информации и гораздо лучше соответствуют природе человеческого восприятия, чем методы традиционного, текстового программирования. В учебнике большое внимание уделяется методам анализа корректности выбранных схем организации параллельных вычислений, включая поиск критических данных, исследование корректности механизмов синхронизации параллельных процессов, поиск тупиковых ситуаций в программе и т.д.

Авторы считают своим приятным долгом выразить признательность профессору Нижегородского государственного университета В.П. Гергелю за конструктивные обсуждения проблем визуального моделирования параллельных вычислений, что вдохновило авторов на написание этой книги.

Неоценимую помощь при подготовке учебника оказали аспиранты кафедры программных систем Д.А. Попова-Коварцева и П.В. Аболмасов, участвующие в написании разделов 4.4 и 4.5 данной книги. Настоящее издание было подготовлено благодаря неизменной дружеской поддержке декана факультета информатики Самарского государственного аэрокосмического университета Э.И. Коломийца.

ВВЕДЕНИЕ

Визуальное программирование – это способ образного графического представления разрабатываемого алгоритма, который более естественен для восприятия человека. При этом существенно уменьшается количество вынужденных ошибок в управлении алгоритма программы, а следовательно, повышается её качество.

Технология графосимволического программирования GRAPH, созданная на кафедре ПС СГАУ, является одним из способов наглядного представления алгоритмов программы в виде графа управления.

Разработка параллельных алгоритмов в корне отличается от традиционного программирования. В настоящее время для организации параллельных вычислений в большинстве случаев используется система MPI. Необходимость равнозначного владения технологией программирования на языках высокого уровня (C++, C#) и MPI ещё дальше отдаляет «конечных» пользователей – специалистов в предметных областях, потребителей суперкомпьютерных технологий. В результате падает эффективность разрабатываемых приложений.

Технология ГСП и созданный на её основе язык визуального представления параллельных алгоритмов имеют большое практическое значение, поскольку позволяют упростить и ускорить разработку алгоритмов параллельных вычислений, предоставляя интуитивно понятное средство описания вычислительных алгоритмов и анализа корректности используемых алгоритмических конструкций.

Данная технология предоставляет пользователям широкие возможности анализа параллельной структуры алгоритма, корректности применяемых механизмов синхронизации вычислений, поиска тупиковых ситуаций, оценки эффективности алгоритма и т.д.

ГЛАВА 1. ОБЗОР МЕТОДОВ И СРЕДСТВ ВИЗУАЛЬНОГО ПРОГРАММИРОВАНИЯ И АВТОМАТИЗАЦИИ ПРОГРАММИРОВАНИЯ

1.1. Методы и средства автоматизации программирования

Широкое внедрение средств вычислительной техники в различные сферы жизни и деятельности человека стимулировало развитие автоматизированных методов и инструментальных средств, применяемых для создания прикладного программного обеспечения (ПО). Производство современного ПО происходит на фоне высоких требований, предъявляемых к качеству создаваемых программ, и значительной сложности выполняемых ими функций. «Идеальной» технологией программирования представляется такая технология, которая по некоторому достаточно неформальному описанию объекта программирования автоматически генерирует текст синтаксически и семантически корректной программы.

Первые попытки «наведения порядка» в области программирования восходят к 70-м годам прошлого века, когда усилиями рабочей группы Технического комитета по программированию Международной федерации по обработке информации (ИФИП), состоящей из таких известных ученых, как Н. Вирт, Д. Грис, Э. Дейкстра, У. Дал, Д. Парнас, Ч. Хоар (руководимой профессором В. Турским), формируются основы методологии и теории программирования.

В среде программистов утверждается *принцип структурного программирования* как наиболее производительного и привлекательного стиля программирования. Развитие концепции *структуризации* привело к осознанию необходимости структуризации данных, что и определило появление в языках программирования *механизмов абстрагирования типов данных* (Клу, Модула и д.р.). Последнее легло в основу *техники модульного программирования*, содержательной основой которой является интерпретация типа как алгебры над множеством объектов и множеством операций, а модуля – как программного эквивалента абстрактного типа.

Эволюция техники модульного программирования привела к появлению объектно-ориентированного стиля программирования, который во многом унифицировал процесс создания ПО. К достоинст-

вам этого метода относится то, что в нем более полно реализуется технология структурного программирования, облегчается процесс создания сложных иерархических систем, появляется удобная возможность создания пользовательских библиотек объектов в различных областях применения.

Параллельно с развитием процедурного стиля программирования в начале 70-х годов появляются непроцедурные языки логического программирования и программирования искусственного интеллекта (LISP, PROLOG, РЕФАЛ, ПРИЗ). Программа в таких языках представляет собой совокупность *правил* (определяющих отношения между объектами) и *цели* (запроса). Процесс выполнения программы трактуется как процесс установления общезначимости логической формулы по правилам, установленным семантикой того или иного языка. Результат вычислений является побочным продуктом процедуры вывода. Такой метод являет собой полную противоположность программирования на каком-либо из процедурных языков. Сегодня PROLOG – язык, предназначенный для программирования приложений, использующих средства и методы искусственного интеллекта, создания экспертных систем и представления знаний.

В 80-х годах XX века исследование причин неудач при реализации больших программных проектов показало, что число ошибок в спецификациях на программы значительно превышает их количество в программных кодах. В это время формулируется тезис о том, что целью программирования является не порождение программы как таковой, а создание технологических условий, когда разрабатываемое программное обеспечение легко адаптируется к новым обстоятельствам и новому пониманию решаемой задачи. Р. Хемминг так формулирует этот тезис: «Здравая вычислительная практика требует постоянного исследования изучаемой задачи не только перед организацией вычислений, но также в процессе его развития и особенно на той стадии, когда полученные числа переводятся обратно и истолковываются на языке первоначальной задачи».

Перечисленные выше причины привели в середине 80-х годов к осознанию необходимости реализации интегрированного окружения поддержки всего жизненного цикла ПО и, в первую очередь, этапа проектирования ПО, что обусловило появление инструментальных средств автоматизации проектирования программных систем (CASE-технологий). Одним из ярких современных представителей средств

автоматизации проектирования ПО является универсальный язык моделирования (Unified Modeling Language)♦. UML создан в результате совместных усилий Г Буча, Д. Рамбо, И. Якобсона, П. Йордана и многих других ученых. В основу языка UML положена система графических нотаций, среди них нотации Г.Буча, технология объектного моделирования OMT, методы объектно-ориентированного проектирования OOSE. В настоящее время язык UML принят как промышленный стандарт многими ведущими производителями программного обеспечения. Следует признать, что язык UML является практически единственным системным средством описания модели проектируемой системы. Он безусловно полезен:

- для формализации требований технического задания на проектируемую систему;
- разрешения (в связи с этим) споров между разработчиком и заказчиком;
- координации взаимодействий коллектива разработчиков.

На ранних этапах проектирования программ Rational Rose может облегчить взаимопонимание заказчика и проектировщика. Не секрет, что программисту иногда приходится объяснять клиенту, что ему нужно. Четкая и аккуратная диаграмма действий легче воспримется заказчиком, чем пространные, полные терминов объяснения отчёта, или бесконечные строки кода программы.

Язык UML предназначен, прежде всего, для разработки программных систем. Его использование особенно эффективно в следующих областях:

- информационные системы масштаба предприятия;
- банковские и финансовые услуги; телекоммуникации;
- транспорт;
- оборонная промышленность, авиация, космонавтика;
- торговые системы;
- медицинская электроника;
- наука;
- распределенные Web-системы.

♦ Богс У., Богс М. UML и Rational Rose 2002. – М.: Лори. 2004. – 509 с.

Сфера применения UML не ограничивается моделированием программного обеспечения. Его выразительность позволяет моделировать, например, документооборот в юридических системах, структуру и функционирование системы обслуживания пациентов в больницах, осуществлять проектирование аппаратных средств.

Применение CASE-инструментов позволяет в значительной степени снизить трудоемкость создания ПО, а в отдельных случаях заменить программирование автоматическим синтезом программ.

Таким образом, развитие методов автоматизации разработки ПО происходит на различных основах (модульное программирование, объектно-ориентированный подход, логическое программирование, CASE-технологии), которые так или иначе развивают концепции структуризации в программировании. Структуризация способствует проведению эффективной декомпозиции проекта, что позволяет получать как целостное представление о ПО, так и его деталях. Однако, несмотря на многочисленные разработки в этой области, в целом проблема автоматизации разработки ПО остается нерешенной по многим причинам как методологического, так и практического характера.

1.2. Методы и средства визуального программирования

В настоящее время отмечается возрастание роли визуализации в используемых технологиях программирования, особенно для коммерческих приложений, связанных, например, с обработкой видеоизображений.

Новый графический подход к решению проблемы автоматизации разработки ПО, основанный на идее привлечения визуальных форм представления программ, в большей степени соответствует образному способу мышления человека. Применение графических методов обещает кардинально повысить производительность труда программиста. Визуальное программирование, бесспорно, обладает достоинством наглядного представления информации и гораздо лучше соответствует природе человеческого восприятия, чем методы традиционного, текстового программирования. Кроме того, графическая форма записи по сравнению с текстовым представлением программ обеспечивает более высокий уровень их структуризации, соблюдение

технологической культуры программирования, предлагает более надежный стиль программирования [4, 11].

ОПРЕДЕЛЕНИЕ. *Визуальным программированием* будем называть процесс графического представления программы с помощью стандартного набора графических элементов.

Наличие стандарта необходимо для организации общения между пользователями, аналитиками, тестировщиками, программистами, менеджерами и остальными участниками достаточно сложного процесса разработки ПО. Под визуализацией программного обеспечения можно понимать совокупность методик использования графики и средств человеко-машинного взаимодействия, применяемых для лучшего уяснения понятий и эффективной эксплуатации программного обеспечения ЭВМ, а также для спецификации и представления программных объектов в процессе создания программ. К системам визуализации программного обеспечения согласно различным классификациям обычно относят системы визуального программирования и системы визуализации программирования, а также некоторые системы программирования в том случае, если они используют визуальные методы представления образцов вводимой и выводимой информации.

В настоящее время известно большое количество более или менее удачных инструментальных средств визуализации программирования. Прежде всего, к ним относятся визуальные средства разработки ПО (рис. 1.1): Visual PROLOG, Visual FoxPro, Visual Studio, Visual BASIC и т.д. В перечисленных выше средствах процедуре визуализации подвергается главным образом пользовательский интерфейс, использующий оконную технологию организации взаимодействия человека со средой программирования. Здесь часто используются разнообразные выразительные средства выделения цветом различных компонент программы (операторов, переменных, функций и т.п.), образные табличные формы представления разнообразной информации, диаграммы, графики и т.д.

Подобного рода средства визуального программирования обычно решают задачи построения пользовательского интерфейса и упрощения разработки приложения путем замены метода написания программы на метод ее конструирования. Однако визуальное представление алгоритмов программ они не затрагивают.

Рис. 1.1

Непосредственно к визуальному стилю программирования в большей степени относятся так называемые «графические языки» и CASE-средства проектирования программ.

Из многочисленного отряда CASE-средств проектирования программного обеспечения отметим его яркого представителя – язык UML. Модель объекта в UML представляется достаточно большим количеством графических нотаций (рис.1.2):

1. Диаграммы вариантов использования.
2. Диаграммы деятельностей.
3. Диаграммы последовательностей (взаимодействий).
4. Кооперативные диаграммы (вид диаграммы взаимодействий).
5. Диаграммы классов.
6. Диаграммы состояний.
7. Диаграммы компонентов.
8. Диаграммы размещений.

Диаграммы описывают различные аспекты создаваемой системы. В то же время следует отметить, что имеющегося достаточно широкого перечня UML диаграмм не всегда хватает для описания извест-

ных объектов, используемых в современных информационных технологиях (базы данных, XML DTD). Вероятно, скоро появятся диаграммы для моделирования систем искусственного интеллекта, нейросетей и т.д. Поэтому диаграммы, используемые в UML, постоянно пополняются новыми диаграммами, отражающими те или иные аспекты новых объектов информатики.

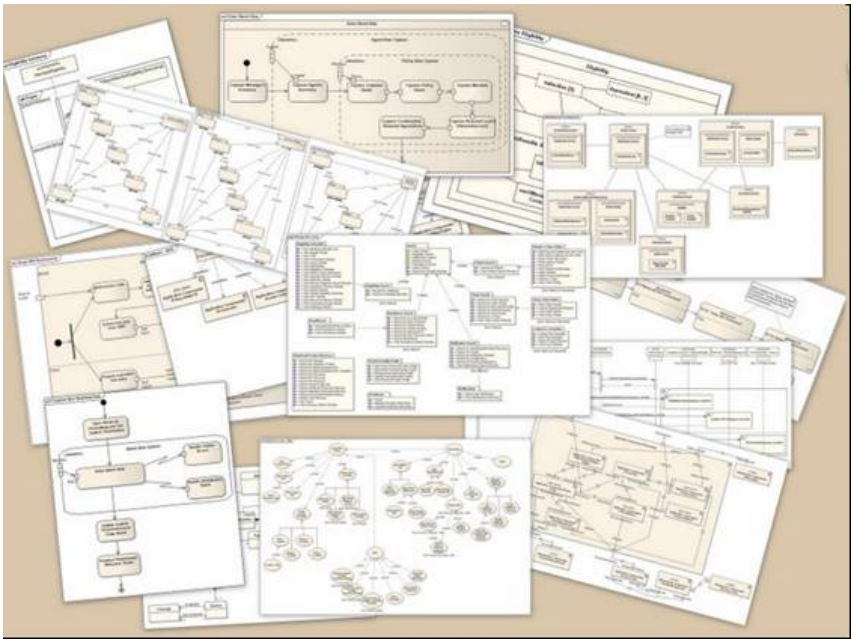


Рис. 1.2. Диаграммы языка UML

Более высокий уровень визуализации достигается в предметно-ориентированных средах программирования. В них визуальные образы полностью соответствуют реальным объектам предметной области, а технология программирования близка к сложившимся традициям, принятым в соответствующей предметной области.

Так, например, для программирования алгоритмов управления технологическими процессами в SCADA системе TRACE MODE 6 [24] поддержаны все 5 языков международного стандарта IEC 61131-3. Среди них есть и визуальные языки – Techno FBD, Techno LD, Tech-

Techno SFC. Такой широкий диапазон средств программирования позволяет специалисту любого профиля выбрать для себя наиболее подходящий инструмент реализации любых задач АСУ ТП и АСУП. Все языки программирования снабжены мощными средствами отладки. SCADA система TRACE MODE 6 обладает великолепной трехмерной графикой. В графике SCADA любой графический элемент может изменять свои свойства, размеры и положение на экране в реальном времени в зависимости от параметров, а также служить кнопкой.

Для удобства редактирования сложных мнемосхем в графическом редакторе SCADA поддерживаются слои, видимостью которых можно управлять. Более того, видимостью слоев можно управлять в реальном времени. Это позволяет на одной мнемосхеме отображать по желанию пользователя те или иные подсистемы технологического объекта. Например, можно создать поверх плана сооружения несколько схем-слоев: канализации, отопления, электроснабжения, вентиляции, пожарной сигнализации и т.д., а отображать на экране дисплея только то, что нужно в данный момент.

В SCADA системе TRACE MODE 6 существенно расширена поддержка внешних графических форматов. Анимация и растровые рисунки могут подвергаться произвольной трансформации (поворот, растяжение), причем не только в редакторе, но и динамически. SCADA TRACE MODE обладает собственным генератором отчетов, позволяющим в реальном времени быстро создавать ясные и полнофункциональные HTML-отчеты. Схематично визуальные возможности системы SCADA представлены на рис. 1.3.

Система **LabVIEW** (Laboratory **V**irtual **I**nstrumentation **E**ngineering **W**orkbench) [23] — это среда разработки для выполнения программ, созданных на графическом языке программирования «G» фирмы National Instruments (США). LabVIEW используется в системах сбора и обработки данных, а также для управления техническими объектами и технологическими процессами. Идеологически LabVIEW очень близка к SCADA-системам, но в отличие от них в большей степени ориентирована на решение задач не столько в области АСУ ТП, сколько в области АСНИ.

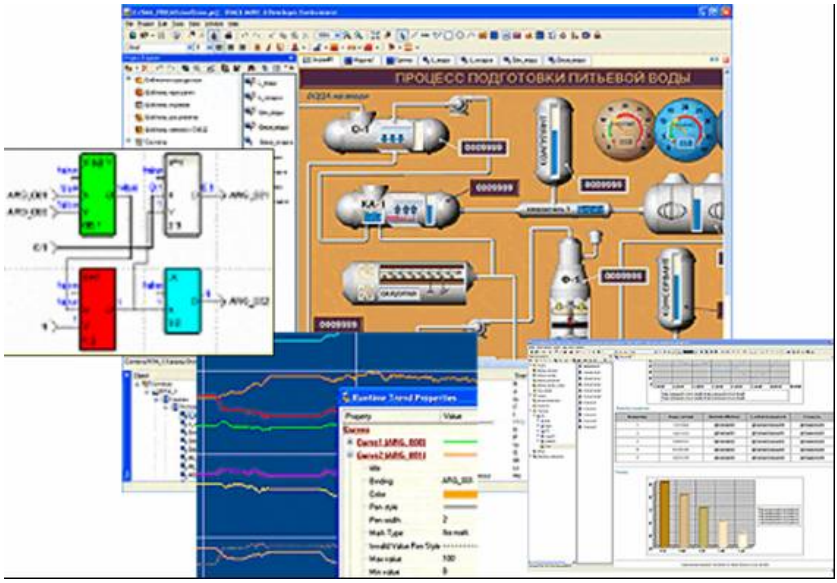


Рис. 1.3. Визуальные образы системы SCADA

Графический язык программирования «G», используемый в LabVIEW, основан на архитектуре потоков данных. Последовательность выполнения операторов в таких языках определяется не порядком их следования, а наличием данных на входах этих операторов. Операторы, не связанные по данным, выполняются параллельно в произвольном порядке.

Схематично визуальные возможности системы LabVIEW представлены на рис. 1.4.

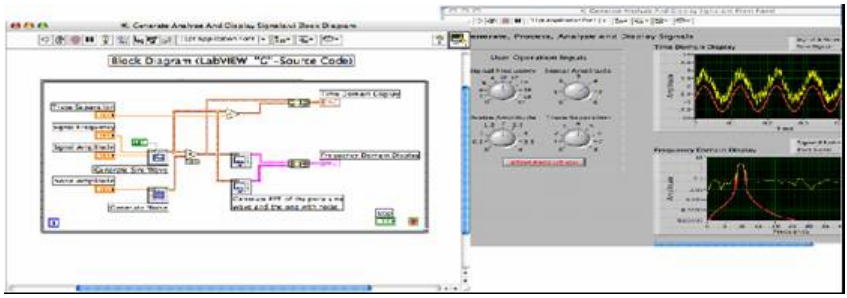


Рис. 1.4. Визуальные образы системы LabVIEW

Большими выразительными визуальными возможностями обладает пакет приложений к MATLAB – SIMULINK [13]. Это интерактивная среда для моделирования и анализа широкого класса динамических систем, в которых программирование реализовано в геометрической форме – в виде блок-диаграмм. Элементами схем являются блоки. Блоки могут быть взяты из библиотек (механика, электротехника, гидравлика и т.п.) или созданы из кода на C, Fortran, Ada. Данный пакет широко используется для проектирования систем управления, цифровой обработки сигналов, коммуникационных систем, имитационного моделирования и т.д. Схематично визуальные возможности системы SIMULINK представлены на рис. 1.5.

Перечисленные системы, конечно, не исчерпывают существующие на данный момент времени проблемно-ориентированные среды программирования. Можно было бы привести примеры визуальных языков имитационного моделирования, пакеты построения нейросетевых приложений, системы проектирования газотурбинных двигателей и т.д. В данном разделе мы остановились на наиболее ярких и широко используемых представителях визуальных проблемно-ориентированных систем программирования.

Что же касается визуализации программирования параллельных вычислений, то эта проблема требует отдельного рассмотрения и будет представлена в следующем разделе.

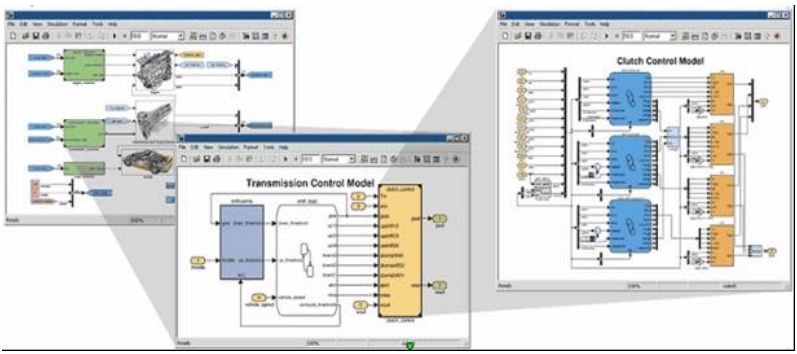


Рис. 1.5. Визуальные образы пакета SIMULINK

1.3. Методы и средства визуального параллельного программирования

Работы в области моделирования и построения параллельных вычислительных процессов можно разделить на два больших направления:

1. Неявный параллелизм. Это направление изучает *методы автоматической генерации параллельных вычислительных процессов* на основе их последовательных прототипов (автоматического распараллеливания последовательных вычислительных процессов).
2. Явный параллелизм. Разработка методов организации вычислений, изначально ориентированных на реализацию на ЭВМ с параллельной архитектурой.

Исследования в области *автоматического распараллеливания* вычислительных процессов необходимы в связи с наличием большого объема ранее разработанных методов, алгоритмов и программ для решения различных задач на последовательных ЭВМ. Их реализация на параллельных ЭВМ требует модификации, связанной с распределением данных и вычислений по узлам параллельной ЭВМ, а также с адаптацией под особенности архитектуры конкретной ЭВМ. Этот процесс важно автоматизировать, чтобы максимально сократить его длительность и избавить исследователей – специалистов в различных предметных областях (которые часто являются авторами и пользователями вычислительных программ) от знакомства со спецификой конкретной ЭВМ, на которой программа будет исполняться.

Автоматическое распараллеливание имеет большое значение и при создании новых вычислительных программ. Последовательные алгоритмы удобны и естественны для человека в силу того, что люди привыкли думать и действовать последовательно. Вместе с тем, любая современная ЭВМ обладает определенной степенью параллелизма. Появление многоядерных процессоров в персональных ЭВМ позволяет с уверенностью говорить о том, что в ближайшем будущем все ЭВМ будут параллельными. В подобных условиях массовое использование ЭВМ для выполнения пользовательских программ, большинство из которых являются последовательными, становится невозможным без применения методов автоматического распараллеливания.

Различают распараллеливание на уровне выражений и распараллеливание на уровне команд. В первом случае используются свойства математических и логических операций (ассоциативность, коммутативность, дистрибутивность) для изменения порядка вычислений в выражениях. Если в результате такого изменения удается выделить независимые друг от друга фрагменты выражения, то их вычисляют одновременно (параллельно).

При распараллеливании на уровне команд исследуются зависимости между различными участками вычислительного процесса с точки зрения используемых данных и очередности выполнения. Независимые друг от друга участки программы выполняются параллельно.

Работы в области автоматического распараллеливания вычислительных процессов принадлежат таким ученым, как С.М. Абрамов [8], В.В. Воеводин, Вл.В. Воеводин [1, 2]. Обычно объектами автоматического распараллеливания являются программы, составленные на одном из универсальных языков программирования.

Отдельной областью исследований в этом направлении является так называемое *функциональное программирование*, в котором «единственным действием является вызов функции, единственным способом расчленения программы на части является введение имени для функции и задание для этого имени выражения, вычисляющего значение функции, а единственным правилом композиции — оператор суперпозиции функции». Функциональное программирование позволяет описать вычислительный процесс в декларативном стиле, который, в отличие от «традиционного» императивного стиля написания программ, предоставляет большую свободу для модификации программы на этапе ее трансляции в исполнимый код с целью распараллеливания. Первыми средствами создания параллельных вычислительных процессов на основе функциональных программ были программные системы: FP, Haskell [6], ПРИЗ. Среди современных систем следует отметить Т-систему [17], ПИФАГОР [26], NUT и т.д.

Наряду с несомненными достоинствами, такими как возможность использования ранее разработанных и хорошо отлаженных последовательных программ, сохранение привычного для человека последовательного стиля разработки вычислительных процессов, обеспечение переносимости программ, *автоматическое распараллеливание* обладает недостатками. Основным недостатком является ограничен-

ность области применения. К сожалению, не все последовательные алгоритмы допускают эффективное распараллеливание. Иногда сам численный метод, на основе которого построен алгоритм, не допускает распараллеливания.

Для достижения максимальной производительности ПО необходимо уже на этапе разработки алгоритма учитывать параллелизм и явно выделять участки, которые должны выполняться одновременно. Более того, необходимо учитывать архитектуру и особенности конкретной параллельной ЭВМ, на которой будет исполняться программа.

Основные сложности, с которыми сталкиваются исследователи в области построения параллельных вычислительных процессов с *явным параллелизмом*, в первую очередь связаны с наглядностью представления вычислительного процесса.

Текстовая нотация, традиционно используемая в математике и программировании, удобна для представления последовательных процессов. Однако последовательная природа самого текста значительно затрудняет восприятие текстового описания параллельных вычислений. На первый план выдвигаются графические способы описания параллелизма.

При разработке визуальных языков параллельного программирования в качестве основного подхода используется рисование графов, как правило, отображающих либо *поток управления*, либо *поток данных*. При этом в графических нотациях используются диаграммы и схемы, зачастую заимствованные из «бумажных» технологий программирования, или специально придуманные для данного случая иконические знаковые системы.

Основой для подавляющего количества графических способов представления параллельных процессов является форма представления в *виде графа*, то есть совокупности вершин (узлов), соединенных между собой дугами (ребрами). В отличие от текстовой формы записи, в которой объекты (символы и слова) образуют *последовательность*, а каждый объект связан только с левым и правым «соседом», графовая форма позволяет наглядно изображать более сложные *взаимосвязи*, поскольку в ней каждый объект может соединяться с несколькими другими объектами. В этом смысле **текстовая форма одномерна**, в то время как **графовая форма – многомерна.** Возможность варьировать геометрические размеры, форму и цвет вер-

шин, внешний вид и толщину дуг, изменять взаимное расположение вершин без изменения топологии графа значительно увеличивает выразительные возможности графовой формы представления алгоритма программы.

Графические модели обычно представляются ориентированными графами, в которых дуги определяют направление передачи данных или иерархию отношения зависимостей вершин друг от друга. Вершины и дуги снабжаются текстовыми аннотациями, которые именуют их, перечисляют их содержимое или свойства.

Различные графические нотации отличаются друг от друга семантикой, вкладываемой в определения вершин и дуг графа. Основные классы графических моделей параллельных процессов приведены на рис. 1.6.

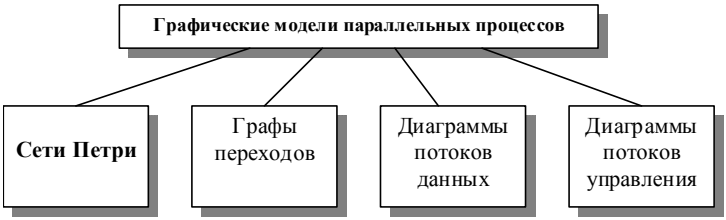


Рис. 1.6. Основные классы графических моделей параллельных вычислительных процессов

Существуют примеры систем (**сети Петри**), где программа на визуальном языке состоит из потока управления, потока данных и множества интерпретаций узлов (то есть процедур, запускаемых при достижении узла). Граф потока управления языка в этом случае обеспечивает представление последовательных потоков, переключение управления, параллелизм и синхронизацию. Параллелизм поддерживается в основном за счет описания того, как несколько маркеров, описывающих поток данных, могут распространяться по графу. Нередко на язык и систему программирования могут возлагаться задачи обеспечения правильности (в частности детерминированности) и эффективности разрабатываемых параллельных программ уже на этапе проектирования и «визуального кодирования».

Возможность формального математического описания сетей Петри в терминах теории множеств, а также достаточно высокий

уровень абстрагирования сетей Петри от деталей и особенностей моделируемого процесса определили их широкую популярность для моделирования дискретных динамических систем. Разработаны и подробно изучены свойства множества подклассов сетей Петри, адаптированных к описанию разнообразных классов сложных систем.

Недостатком сетей Петри при описании вычислительных процессов является необходимость перехода к специфическим терминам, используемым в сетях Петри, таким как переходы, места, разметки. Такое преобразование не всегда очевидно. Кроме того, представить процесс функционирования сети Петри можно только в динамике срабатываний её переходов, например, используя возможности анимации графических объектов. В то время как статическое изображение сети обладает невысокой наглядностью. В качестве примера использования сетей Петри на рис. 1.7 представлена параллельная модель суммирования элементов матрицы с заданным числом столбцов и произвольным числом строк.

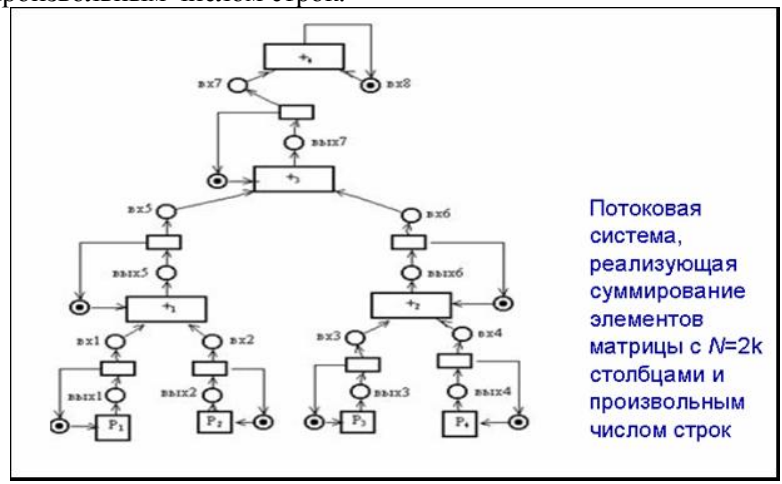


Рис. 1.7. Пример Сети Петри

Графы переходов (диаграммы состояний, диаграммы изменений состояний) представляют собой графическую форму описания конечных автоматов. Вершины графа переходов соответствуют состояниям автомата, а дуги определяют возможные переходы из одного состояния в другое. Традиционно графы переходов используются для

описания «последовательных» автоматов, так как одновременные переходы по двум и более дугам, исходящим из одной вершины, считаются запрещенными. Если такие переходы разрешены, то графы переходов, в которых допустимо одновременное существование нескольких «активных» вершин, называются *графами переходов с параллелизмом*. Графические модели на основе графов переходов используются в SWITCH-технологии [5] и графическом языке Statecharts.

В Институте кибернетики имени В.М. Глушкова ещё в 70-х годах XX века была разработана Р-технология программирования, использующая графическую модель описания алгоритмов, близкую к автоматным, и получившую название **Р-схем** [11]. Р-схема – нагруженный по дугам ориентированный граф, изображаемый с помощью горизонтальных и вертикальных линий и состоящий из структур с одним входом и одним выходом. Построение Р-схем осуществляется из базовых графических структур двух видов, представленных на рис. 1.8,а.

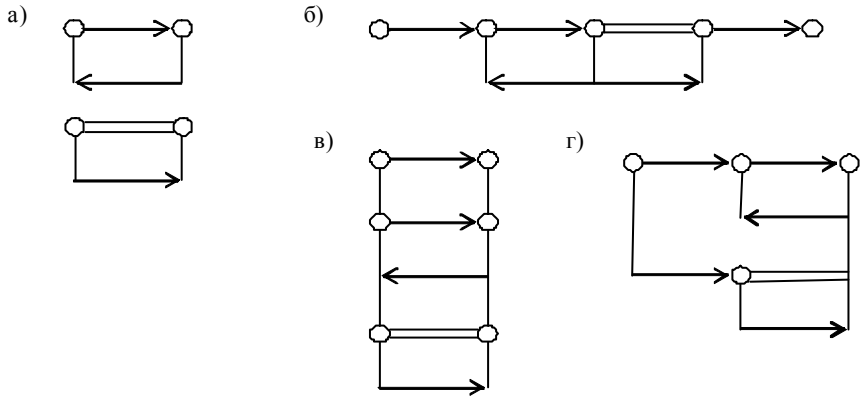


Рис. 1.8. Пример построения Р-схем

Остальные конструкции получаются с помощью операций последовательного, параллельного и вложенного соединения базовых элементов. Примеры таких конструкций приведены на рис. 1.8, б, в и г соответственно. Для Р-схем разработан и введен в действие ГОСТ 19.005-85, определяющий внешнюю форму их представления.

В Р-схемах используются два принципиально различных алфавита для задания логики алгоритма (графический алфавит) и функциональных операторов, связанных с обработкой данных (текстовый алфавит). Задаваемый Р-схемой ориентированный граф заменяет собой операторы управления, представляемые в текстовых языках программирования инструкциями if, while, repeat, case и др. Направленная дуга изображает переход из одного состояния вычислительного процесса в другое. Над дугой записывается логическое условие прохождения по дуге, а запись под дугой относится к функциональной составляющей и определяет выполняемые при проходе по дуге действия по обработке данных. Текстовые записи осуществляются на одном из традиционных языков программирования, например С или Паскале.

В ранних визуальных языках при проектировании программы на графовых схемах могла отображаться потенциальная возможность параллельного выполнения кода. Обычно в этих языках параллельные дуги графа описывали потенциально параллельные участки программы. При этом реальное создание эффективного параллельного кода возлагалось не на программиста, а на систему программирования. На следующем этапе развития визуальных языков параллельного программирования графы используются уже для создания настоящей параллельной программы с описанием взаимодействия ее процессов. Во многих случаях такие языки базируются на **диаграммах потоков данных**.

На таких диаграммах можно образно изобразить процессы порождения данных одним из узлов схемы, реализующего последовательные вычисления для нужд других процессов. Параллельность обычно задается дополнительной параллельной дугой. При программировании вначале рисуется граф потоков данных решения данной задачи, а на следующем этапе за счет текстовых описаний портов ввода и вывода процесса аннотируются узлы графа (последовательные процессы), а также пишутся правила запуска этих узлов.

Диаграмма потоков данных позволяет наглядно изобразить интерфейс по данным между различными участками вычислительного процесса. Множество входящих дуг каждой вершины определяет ее входные данные. Результаты вычислений, производимых в вершине, передаются по исходящим дугам другим вершинам. Диаграммы по-

токов данных используются, например, в системах CODE 2.0, Paralex.

Диаграммы потоков управления описывают передачу управления между компонентами вычислительного процесса. Модель в этом случае также представляется ориентированным графом, в котором дуги определяют порядок запуска на исполнение вершин графа. Каждая вершина соответствует некоторой последовательности вычислительных операций или другому графу, если модель допускает иерархию. Простейшим примером таких моделей являются блок-схемы. Достоинством диаграмм потоков управления является наглядность описания вычислительного процесса, а основным недостатком – непрозрачность интерфейса по данным между вершинами. Примерами моделей этого типа являются графический язык системы HeNCE [25], технология графосимволического программирования [4].

Специфической задачей при работе с параллельными вычислениями является обеспечение согласованной работы параллельных вычислительных процессов (синхронизация). Согласованность работы заключается, во-первых, в корректном использовании множества данных, с которыми одновременно работают несколько вычислительных процессов, а во-вторых, в беспрепятственном выполнении параллельных вычислительных процессов, при наличии их взаимного влияния друг на друга.

В последовательном вычислительном процессе в каждый момент времени над любой переменной может выполняться только одна операция, определенная алгоритмом. При параллельных вычислениях одновременно выполняются несколько вычислительных процессов, поэтому возможна ситуация, когда к одной и той же переменной в некоторый момент времени обращаются два или более процессов. При этом один процесс может считывать значение переменной, а другой – изменять его. Если моменты чтения и изменения значения переменной не согласованы между собой, то результат операции чтения непредсказуем.

Для исключения подобных ситуаций разработаны различные механизмы синхронизации параллельных вычислительных процессов: критические секции, семафоры, мониторы, сообщения.

1.4. Краткий обзор раздела

Современное программное обеспечение характеризуется высокой сложностью выполняемых функций и значительностью объемов исполняемых кодов. При этом к ПО предъявляются высокие требования по качеству и надежности ее кодов. За сравнительно небольшой промежуток времени методы и средства разработки ПО прошли эволюционный путь развития, сравнимый с методами проектирования сложных технических систем, что обуславливает необходимость обеспечения максимальной эффективности процессов проектирования, разработки, эксплуатации и сопровождения ПО на всех этапах жизненного цикла.

Актуальным направлением сокращения сроков проектирования, разработки, модификации, сопровождения ПО и повышения качества программ является разработка технологических основ комплексной автоматизации перечисленных процессов в течение всего жизненного цикла.

Одним из методов повышения производительности труда программистов является применение графических методов программирования, обеспечивающих более высокий уровень структуризации ПО. Графическая нотация является более наглядной и компактной по сравнению с текстовым описанием. За счет использования графических моделей удается не только сократить время разработки параллельных вычислительных процессов, но и повысить их надежность, т.к. графическая нотация допускает формальное математическое описание модели, по которому может быть проведена ее автоматическая верификация и оптимизация.

Весьма значительная интеллектуальная сложность параллельного программирования настоятельно требует инструмента представления как абстракций, так и программных объектов на всех этапах разработки параллельных программных комплексов.

1.5. Контрольные вопросы

1. Какие средства автоматизации проектирования программного обеспечения Вы знаете?
2. Перечислите основные парадигмы программирования, способствующие развитию методов автоматизации разработки программного обеспечения.

3. Что такое визуальное программирование?
4. Классификация средств визуального программирования.
5. Какие визуальные средства разработки программного обеспечения Вы знаете?
6. Перечислите графические языки визуального программирования.
7. Какие основные направления разработки программ, описывающих параллельные вычислительные процессы, Вы знаете?
8. Опишите основные классы графических моделей параллельных вычислительных процессов.
9. Какие графические нотации используются в средстве визуального описания моделей параллельных процессов в сетях Петри?
10. Содержательное наполнение вершин и дуг в диаграммах описания потоков данных.
11. Содержательное наполнение вершин и дуг в диаграммах описания потоков управления.

ГЛАВА 2. ТЕХОЛОГИЯ ГРАФОСИМВОЛИЧЕСКОГО ПРОГРАММИРОВАНИЯ

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

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

Однако, несмотря на наличие развитых визуальных средств представления информации, стиль программирования в таких системах по-прежнему остается текстовым, если не считать так называемых «визардов», предназначенных для автоматизации отдельных технологических операций, связанных с генерацией программных приложений.

Создание визуального (образного) стиля разработки ПО является основным мотивом разработки технологии графосимволического программирования.

2.1. Концептуальная модель ГСП

2.1.1. Основные положения

ОПРЕДЕЛЕНИЕ. *Технологию ГСП* определим как технологию проектирования и кодирования алгоритмов программного обеспечения, базирующуюся на графическом способе представления программ, преследующую цель полной или частичной автоматизации процессов проектирования, кодирования и тестирования ПО.

Данная технология программирования исповедует два основополагающих принципа:

- *визуальную, графическую форму* представления алгоритмов программ и других компонент их спецификаций;
- *принцип структурированного процедурного программирования*.

В качестве методологической основы для представления алгоритмов в работе используется модель объекта с дискретными состояниями [4]. Основу такой модели составляет предположение о том, что для любого объекта программирования тем или иным способом можно выделить конечное число состояний, в которых он может пребывать в каждый момент времени. Тогда развитие вычислительного процесса можно ассоциировать с переходами объекта из одного состояния в другое. В математике такая концепция в качестве способа абстрагирования плодотворно используется достаточно давно: марковские цепи, теория массового обслуживания, теория формальных грамматик и автоматов, моделирование систем и т.д.

Для уточнения понятия *состояния*, используемого в работе, определимся с принятой в технологии графосимволического программирования [4] концепцией модели алгоритма. Можно выделить следующие три основных типа универсальных алгоритмических моделей.

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

Второй – основан на представлении об алгоритме как о некотором детерминированном устройстве, способном выполнять в каждый

отдельный момент лишь примитивные операции. Одним из многочисленных представителей этого типа является *машина Тьюринга*.

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

Для технологии графосимволического программирования наиболее подходящим является первый тип формализации понятия алгоритма[1], когда произвольная программа интерпретируется некоторой вычислимой функцией:

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

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

ОПРЕДЕЛЕНИЕ. Определим *граф состояний G* как ориентированный помеченный граф, вершины которого – суть состояния, а дугами отмечены переходы системы из одного состояния в другое.

Каждая вершина графа помечается соответствующей локальной вычислимой функцией f_k. Одна из вершин графа, соответствующая начальному состоянию, объявляется *начальной вершиной* и, таким образом, граф оказывается *инициальным*. Дуги графа проще всего интерпретировать как *события*.

---

[1] Более строго данная концепция определения алгоритма исследована для вычислительной модели, названной *машиной Колмогорова* [18]. Там же можно найти едва ли не единственное формальное определение понятия *состояния* вычислительного процесса. По Колмогорову *состояние* – суть, достигнутая конструктивным объектом в некоторый момент времени, определённая конкретизация структур данных (входных или вычисляемых). Тогда множество состояний – ансамбль конкретизаций структур данных. На каждом шаге итерации реализуется переработка текущего состояния структур данных D в новое состояние D^* с помощью некоторой локальной функции $D^* = A_k(D)$. Процесс переработки $D^0 = D$ в $D^1 = A_{k_1}(D^0)$, D^1 в $D^2 = A_{k_2}(D^1)$ и т.д. продолжается до тех пор, пока не появится сигнал о получении решения.

ОПРЕДЕЛЕНИЕ. В технологии ГСП под с*обытием* понимается любое изменение *состояния* объекта ***O***, влияющее на развитие вычислительного процесса.

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

Для реализации *событийного управления* на графе состояний ***G*** введем множество предикативных функций $P = \{ P_1, P_2, ..., P_l \}$.

ОПРЕДЕЛЕНИЕ. Под *предикатом* будем понимать логическую функцию $P_i(D)$, которая в зависимости от достигнутых значений данных D принимает значение, равное 0 или 1.

Дугам графа ***G*** поставим в соответствие предикативные функции. Событие, реализующее переход $S_i \to S_j$ на графе состояний ***G,*** инициируется, если модель объекта **O** на текущем шаге работы алгоритма находится в состоянии S_i и соответствующий предикат $P_{ij}(D)$ (помечающий данный переход) истинен.

В общем случае предложенная концепция (без принятия дополнительных соглашений) допускает одновременное наступление нескольких событий, в том случае, когда несколько предикатов, помечающих дуги (исходящих из одной вершины), приняли значение истинности.

Возникает вопрос: на какое из наступивших событий объект программирования должен отреагировать в первую очередь?

Традиционное решение этой проблемы связано с использованием механизма приоритетов. В связи с чем все дуги, исходящие из одной вершины, помечаются различными натуральными числами, определяющими их приоритеты. Отметим, что принятое уточнение обусловлено ресурсными ограничениями, свойственными однопроцессорной ЭВМ.

Существенно, что изображение программ в виде ориентированного помеченного графа, естественно для восприятия человеком. Направленная дуга служит очевидным изображением перехода из одного состояния вычислительного процесса в другое, вершина – выполняемой вычислительной функции, а в целом ориентированный граф наглядно представляет все пути, по которым может развиваться вычислительный процесс. В этом случае логические особенности разрабатываемого программного модуля проявляются в характерной для него топологии графа. Можно сказать, что графическое представление программ позволяет задействовать непосредственное образное восприятие, расширяя возможности человека при разработке и анализе сложных программ.

ОПРЕДЕЛЕНИЕ. Определим *универсальную алгоритмическую модель* технологии графосимволического программирования четверкой

$$M=<D,\ \mathfrak{J},\ P,\ \boldsymbol{G}>, \qquad (2.1)$$

где D – множество данных (ансамбль структур данных) некоторой предметной области программирования; $\mathfrak{J}$ – множество вычислимых функций некоторой предметной области $f_k(D)\in\mathfrak{J}$; P – множество предикатов, действующих над структурами данных предметной области D; $\boldsymbol{G}$ – граф состояний объекта $\boldsymbol{O}$.

ОПРЕДЕЛЕНИЕ. Под *предметной областью программирования* (ПрОП) будем понимать некоторую среду программирования, имеющую *общую цель* – разработку программного обеспечения автоматизации расчетов в некоторой области практических интересов (авиационные двигатели, бизнес, медицинские приборы и т.д.), *общую область данных* и *общую область знаний*.

Представленная алгоритмическая модель является универсальной, поскольку допускает описание любых алгоритмов.

Таким образом, в технологии ГСП в качестве универсальной алгоритмической модели предлагается использовать абстрактную модель $M=<D,\ \mathfrak{J},\ P,\ G>$, основанную на графе состояний.

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

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

2. Происходит декомпозиционное расслоение основных компонент описания алгоритма программного продукта. Так, структура

алгоритма представляется графом ***G*,** элементы управления собраны во множестве предикатов *P* и, как правило, значимы не только для объекта ***O***, но и для всей предметной области. Спецификация структур данных, а также установка межмодульного информационного интерфейса по данным «пространственно» отделена от описания структуры алгоритма и элементов управления.

Предложенная алгоритмическая модель $M = <D, \Im, P, G>$ в конечном счете описывает некоторую вычислимую функцию $f_G(D)$ и в этом смысле может служить «исходным материалом» для построения алгоритмических моделей других программ. Последнее означает, что технология ГСП допускает построение иерархических алгоритмических моделей. Уровень вложенности граф-моделей в ГСП не ограничен.

Структура алгоритмической модели $M = <D, \Im, P, G>$ во многом зависит от выбранного способа декомпозиционного расслоения объекта программирования на множество состояний *S* и множество событий, определяемых предикативными функциями *P*. В каждой конкретной предметной области эта задача решается индивидуально и, как правило, не вызывает затруднений.

2.1.2. Онтологии в технологии ГСП

В последнее время в области информационных технологий широко используется онтологический подход к описанию свойств объектов исследования. В этом смысле представленная технология ГСП по своему специфицирует, структурирует и накапливает знания об объекте исследования по мере его разработки.

Основоположник использования онтологий в области информационных технологий Томас Груббер определяет онтологию как «спецификацию концептуализации». При этом под «концептуализацией» можно понимать спецификацию знаний об окружающем мире, т.е. описание структуры Бытия безотносительно к какой-либо инженерной задаче. Для программистов естественнее и ближе понимание концептуализации как построение модели решаемой задачи, т.е. её концептуальной схемы.

ОПРЕДЕЛЕНИЕ. Под *формальной моделью онтологии* ***O*** часто понимают упорядоченную тройку вида

$$O =< C, R, F >, \qquad (2.2)$$

где C – конечное множество понятий (концептов) предметной области, R – конечное множество отношений между понятиями предметной области, F – конечное множество функций интерпретации, заданных на понятиях и/или отношениях онтологии $\boldsymbol{O}$.

Сравним между собой концептуальные схемы (2.1) и (2.2). Множество понятий (концептов) предметной области C онтологии $\boldsymbol{O}$ можно связать с множеством состояний S модели M объекта О. Конечное множество отношений R онтологии $\boldsymbol{O}$ в технологии ГСП описывается графами состояний G_i. И, наконец, конечное множество функций интерпретации F – это множество вычислимых функций F и предикатов P.

Фактически модель алгоритма (2.1) *содержит описание онтологии* разрабатываемого объекта О в данной предметной области. А в совокупности все модели объектов предметной области формируют описание её онтологии. Как правило, множество предикатов и, в меньшей степени, множество вычислимых функций общезначимы для всей предметной области и повторно используются в разных объектах ПрОП.

Данное обстоятельство наделяет технологию ГСП рядом полезных свойств. Кроме удобной для программиста визуальной формы описания модели алгоритма (данные аспекты были уже описаны выше) оказалось, что сформированная средствами ГСП предметная область содержит всю необходимую информацию для автоматической генерации отчетов, разрабатываемых программных приложений. Но главной особенностью данной технологии является возможность построения модели алгоритма без заранее сформировавшейся структуры алгоритма.

В процедурных языках программирования для написания программы решения некоторой практической задачи необходимо первоначально решить задачу, т.е. разработать алгоритм, а затем реализовать его в программных кодах. В технологии ГСП для разработки алгоритма, а следовательно и кодов программы, достаточно иметь лишь идею решения задачи.

Последнее связано, по-видимому, с присутствием онтологических аспектов в технологии ГСП. При разработке программных приложений в технологии ГСП пользователь описывает онтологию

предметной области, вводя для неё новые данные, новые понятия и функциональные отношения по мере изучения объекта программирования, в итоге параллельно формируется и модель алгоритма программы, связывающая все эти понятия.

Отметим ещё одно обстоятельство. Обычно онтологии используются в так называемых интеллектуальных системах (ИС) в интересах обеспечения взаимодействия ИС между собой и с человеком. Здесь онтологии рассматриваются как интерфейсы интеллектуальных систем. И как следствие, возникают задачи разработки стандартов представления данных, процедурных и декларативных знаний, языков описания знаний, а также методов и средств слияния онтологий.

С точки зрения программиста, онтология напоминает систему управления базой данных, с той лишь разницей, что хранятся не данные, а знания. За построение онтологии отвечают одни программные средства, в то время как использование этих знаний – привилегия других программных приложений. Одни программы хранят знания, другие содержат «движок», позволяющий использовать эти знания, например, делать выводы в экспертных системах, реализовывать интеллектуальный поиск необходимой информации на Web-ресурсах и т. д.

2.2. Базовые модули и типы данных

2.2.1. Типы данных

Понятие *типа данных* T сигнатуры Σ определим как пару:

- *спецификация типа* данных сигнатуры Σ,
- соответствующая ей *реализация типа* данных [14].

Здесь под сигнатурой понимается пара $\Sigma = \langle B, \Omega \rangle$, где B – множество имен-основ (имена базовых типов базового языка программирования или производных от этих типов данных), а Ω есть $\left(B^* \times B\right)$ – индексированное семейство множеств имен операций, B^* – множество всех цепочек элементов множества B.

Например,

$B = \{\text{int}, double, real, \ldots\}$,

$\Omega = \{+(double, double \to double), +(double, real \to double), \ldots\}$.

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

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

В этом смысле чисто синтаксический аспект спецификации типа данных можно считать недостаточным.

В технологии ГСП понятие типа данных расширено понятием интерпретации данного [4]. Действительно, во многих предметных областях чисто «языковое» представление типа данного является неполным. Например, в физике в формуле $F = am$ все три компоненты формулы *F, a, m* имеют естественный языковый тип double с определенными на этом типе операциями (+, -, *, /), однако каждое из перечисленных данных имеет самостоятельную смысловую интерпретацию (для физических параметров она определяется его размерностью): *F* – сила [Н], *a* – ускорение [м/с^2], *m* – масса [*кг*].

Очевидно, что передача в подпрограмму вместо ускорения *a* (тип double) скорости *v* (тип double) не вызовет синтаксической ошибки, но приведет к неправильному исполнению программы. Следует отметить, что такого рода ошибки обычно очень сложно распознаются.

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

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

$$S_{\text{int}} = \{[\text{м}], [\text{кг}], [\text{с}], [\text{м/с}], \ldots\}.$$

Множество операций в простейшем случае может быть представлено естественными операциями над типами интерпретации $\Omega_{\text{int}} = \{+,-,*,/\}$. При описании типов интерпретации данных базовых

модулей необходимо ввести множество аксиом A_{int} , действующих над типами интерпретаций. Множество A_{int} состоит из замкнутых σ - формул. Например, для формулы $F = am$ аксиомой типообразования служит формула: $T_1 = T_2 * T_3$.

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

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

2.2.2. Базовые модули

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

Порождение первоначального множества вычислимых функций (базовых модулей) производится на любом из существующих языков программирования, например, на языке С++.

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

ром, в рамках структурного программирования развивает теорию программирования как методологию понимания сложных проблем. Обилие «технических» трудностей в программировании возникает всякий раз при решении конкретных практических задач, которые по Б.Мейеру [16] связаны одновременно с ограничениями, внутренне присущими человеческому уму, и с общей проблемой передачи идей и разделением труда.

Основной задачей методологии программирования является сведение при помощи систематической декомпозиции задач очень большой сложности к комбинации простых задач, чтобы при этом синтез решений было легко реализовать. Для того чтобы построить программу, необходима стратегия декомпозиции. Однако остается уточнить, каким должен быть основной элемент в построении программы, т.е. единица декомпозиции.

Наиболее простой классический ответ заключается в том, чтобы рассматривать элемент программы, определенный своими входами и выходами, т.е. подпрограмму. Подобная концепция вводит понятие *модуля.* Можно выделить два подхода к интерпретации этого понятия.

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

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

В этом случае модуль превращается в активный элемент (объект), обладающий всем множеством операций (методов) своего класса, а составление программы происходит в терминах взаимодействующих объектов, где активные элементы – именно объекты, а не использующие их процессы. Таким образом, речь идет о необходимости считать программу не совокупностью процессов, обрабатывающих

данные, а совокупностью активных «машин», взаимодействующих между собой. На практике это направление получило распространение в так называемых объектно-ориентированных языках программирования.

Таким образом, перечисленные выше подходы к определению понятия *модуля* порождают две стратегии декомпозиции задачи построения программы и в конечном итоге – альтернативные парадигмы программирования. Интересно, что в объектно-ориентированном программировании методология программирования поставлена во главу угла, поскольку успех программирования в этом стиле во многом зависит от корректности выполнения операции «расслоения свойств» предметной области, для которой разрабатывают программы [12]. В объектно-ориентированном подходе программная таксономия проводится всегда либо осознанно, либо стихийно. Методы структурного программирования настоятельно рекомендуют проводить предварительный модульный анализ предметной области, однако на практике благие намерения редко реализуются.

ОПРЕДЕЛЕНИЕ. Под *модулем* будем понимать независимую программную единицу, реализующую определенную функцию в процессе преобразования некоторого агрегата данных.

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

2.2.3. Типы модулей

ОПРЕДЕЛЕНИЕ. Введем понятие *типа модуля (вычислимой функции)* как обобщение понятия типа операции, которое определим как множество отображений из области определения функции в область её результата.

Причем областью определения функции считается декартово произведение множеств значений нескольких типов данных (типов формальных параметров), областью результатов – множество значений некоторого одного типа данных. При этом тип функции изображается $T_1, T_2, ..., T_n \to T_p$, где $T_1, T_2, ..., T_p$ – типы формальных параметров и результата вычислений рассматриваемой функции.

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

В теории программирования подпрограммы – одно из фундаментальных средств абстрагирования и искусственного расширения возможностей любого языка. Однако правила реализации первой из перечисленных выше операций – составления тела подпрограммы, обычно синтаксически неотличимы от правил кодирования основного текста программы. Последнее достигается за счет введения понятия формальных параметров. В то время как между основной программой и подпрограммами имеет место существенная разница.

При разработке основной программы преследуется конкретная практическая цель, например, произвести расчеты некоторого технического устройства, физического явления, математических формул и т.д. При этом данным, используемым в программе, «придается» вполне конкретный смысл (давление, скорость, пропускная способность и т.п.). В подпрограммах формальные параметры лишены конкретной смысловой нагрузки.

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

В технологии ГСП исходные, модифицируемые данные и результаты вычислений базовых модулей размещаются в списке типов данных реализуемой функции, поэтому тип базового модуля определим

как отображение типов данных из области определения на область их значений $B : T_{i_1}, T_{i_2}, ..., T_{i_n} \rightarrow T_{j_1}, T_{j_2}, ..., T_{j_m}$, которые реализуются в соответствии с принятыми в *B* операциями над типами данных.

2.3. Объекты технологии

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

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

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

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

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

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

2.3.1. Акторы

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

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

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

Между базовым модулем и актором осуществляется односторонняя связь типа «один ко многим». Каждый актор имеет свой прототип в виде базового модуля, а на его основании можно построить один или несколько акторов. Это свойство полиморфизма объектов позволяет избежать избыточности при порождении новых акторов, которые различаются между собой только привязкой к данным. Другими словами, на основе одного отлаженного и оттестированного базового модуля за счет механизма автоматизированной привязки по данным можно построить несколько корректных акторов, что позволяет значительно повысить уровень надежности порождаемых программных кодов.

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

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

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

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

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

1) инициируемые (импортируемые) данные (I), которые должны принять значения до их использования объектом;

2) вычисляемые (экспортируемые) данные (V), которые впервые получают свои значения в процессе выполнения объекта;

3) модифицируемые (изменяемые) данные (M), которые образуются путем пересечения множеств инициируемых и вычисляемых данных.

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

<имя объекта>(<указатель на структуру данных ПрОП>).

Произвольный базовый модуль, разработанный на языке C++, например:

```
B(t1, t2,...,tn)
  { тело базового модуля; }
```

в процессе регистрации превращается в модуль стандартного вида:

```
B~(void *G)
  { return(B(<тип t1> G.D1, <тип t2>G.D2, <тип tn>G.Dn));}
```

Для стандартизованной формы базового модуля операция конкретизации типов данных сводится к процедуре формирования списка фактических параметров, который определяется паспортом порождаемого актора. В этом смысле паспорт актора и базовый модуль полностью определяют актор ПрОП. На рис. 2.1. показана схема порождения *акторов* расчета таких узлов авиационного двигателя, как вентилятор и компрессоры высокого и низкого давления, путем привязки типов данных абстрактной вычислительной схемы, «заложенной» в соответствующем базовом модуле.

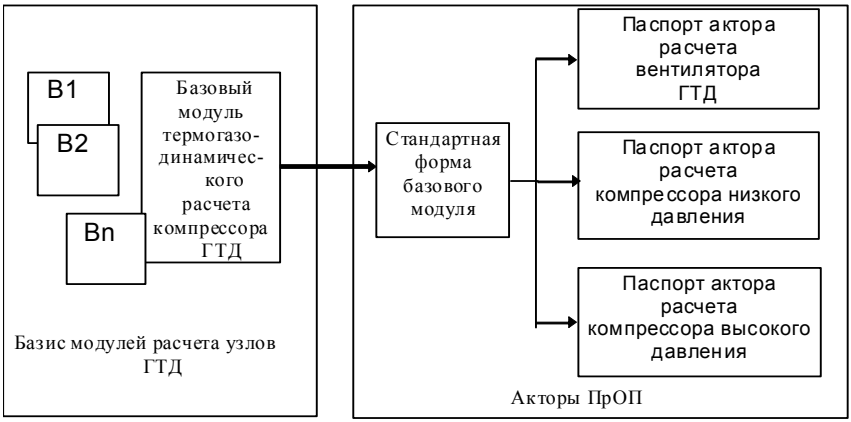


Рис. 2.1. Порождение акторов в предметной области

2.3.2. Предикаты

В процессе реализации алгоритма в рамках технологии ГСП передача управления между объектами осуществляется с помощью управляющих объектов-предикатов. Формально предикат представляет собой отображение из множества данных предметной области на множество логических значений «истина» или «ложь»:

$$P_k : (d_1, d_2, \dots , d_m) \rightarrow \{ 0, 1 \}.$$

Отличие предиката от актора заключается в том, что предикат не может производить преобразование над данными, то есть все его данные являются входными: $(d_1, d_2, \dots , d_m) \in D^{in}$.

Технологически порождение предикатов ничем не отличается от процедуры порождения акторов. Первоначально строится базис абстрактных логических функций, действующих над типами данных. Множество предикатов ПрОП строится из абстрактных логических функций в результате аппликации их типов данных к данным ПрОП, т.е. в результате паспортизации типов данных логических функций. Рассмотрение *акторов* и *предикатов* как различных категорий *объектов* ГСП обусловлено не только особенностями реализуемых ими типов отображений, но и, что особенно важно, их ролевыми назначениями.

Акторы, являясь локальными вычислимыми функциями, реализуют преобразование данных ПрОП, в то время как предикаты, являясь функциями управления вычислениями, образуют базу знаний для всей предметной области, т.е. они общезначимы для любых программ, разрабатываемых в рамках ПрОП.

2.3.3. Агрегаты

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

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

$$G = \{ F, P \},$$

где $F = \{ A_1, A_2, \ldots , A_n \}$ – множество акторов, которые являются вершинами графа, $P = \{ P_1, P_2, \ldots , P_m \}$ – множество предикатов, которые представляют дуги графа.

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

Развитие вычислительного процесса в агрегате происходит путем передачи управления из одной вершины в другую, начиная с корневой. Этот процесс может быть завершен по двум причинам: либо достигнута концевая вершина графа, из которой нет исходящих дуг, либо из текущей вершины отсутствуют разрешенные другими предикатами переходы в другие вершины. Если в процессе передачи управления сложилась ситуация, когда истинными одновременно являются несколько предикатов, то управление будет передано по предикату, имеющему наибольший приоритет.

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

В отличие от акторов и предикатов, которые полностью определяются своими паспортами, при порождении агрегата с помощью

специального компилятора формируется текст нового объекта ПрОП, который после трансляции заносится в библиотеку объектных модулей ПрОП.

2.3.4. Объекты типа «*in line*»

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

2.4. Модель межмодульного интерфейса

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

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

2. Проблемы *распределения и управления* памятью ЭВМ.

3. Проблема *отладки* межмодульного информационного интерфейса, которая связана с декларативным стилем его построения, когда операторы объявления типов данных, самих данных, выделение

оперативной памяти под данные, а также указание областей их действия «размазаны» по тексту программы. В таких условиях идентификация ошибок в информационном интерфейсе превращается в нетривиальную задачу.

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

Анализ ситуации показывает, что значительная часть проблем обусловлена слишком большими возможностями, предоставляемыми в языках программирования для построения межмодульных интерфейсов.

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

Так, например, на языках управления базами данных (например, FOXPRO) обмен информацией между подпрограммами обеспечивается за счет реализации механизма глобального описания переменных, если не учитывать возможность обмена данными через таблицы базы данных.

На языке ПРОЛОГ проблема организации информационного интерфейса вообще отсутствует, поскольку в концепциях языка заложены идеи логического вывода целевых условий в соответствии с аксиоматикой предметной области, а не выполнения процедурных действий над данными. При этом все операции, выполняемые над данными, скрыты (осуществляются автоматически) от пользователя. Вероятно, благодаря именно этой особенности, язык ПРОЛОГ позволяет порождать на редкость надежные программные коды в очень короткие сроки.

В языках, ориентированных на объекты, новая парадигма программирования невольно отделила описание структур данных (классов) от подпрограмм, их использующих (методов). В новых концепциях основное внимание программиста сосредоточено на формиро-

вании модели данных описываемого объекта, чем обеспечивается более высокий уровень надежности разрабатываемых программ. Не менее важен факт автоматического порождения поколений данных при создании новых объектов. Однако дела обстоят благополучно до тех пор, пока построенная «пирамида» классов удовлетворяет поставленным перед программистом целям и задачам. Любые, даже незначительные изменения, например в целевых установках на разрабатываемое программное обеспечение или в моделях используемых структур данных, требуют серьезного изучения иерархии классов, их свойств и методов и в конечном итоге значительных усилий на модификацию пирамиды классов.

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

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

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

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

4. Привязка данных объектов ПрОП реализована в паспортах объектов ПрОП.

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

Предложенный стандарт позволяет полностью отделить задачу построения межмодульного информационного интерфейса от кодирования процедурной части программы, а также частично автоматизировать процессы построения информационного интерфейса.

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

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

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

Формально сущность проблемы организации передачи данных между объектами в рамках некоторого модуля-агрегата f_Σ можно определить как задачу построения области данных агрегата f_Σ – $D_\Sigma = D_\Sigma^{in} \bigcup D_\Sigma^{out}$ и установления соответствий между данными $D_\Sigma = \{d_1, d_2, ...d_{n_\Sigma}\}$ и данными $D_i = \left\{ d_1^i, d_2^i, ..., d_{n_i}^i \right\}$ объектов $f_1, f_2, ..., f_m$, из которых составлен агрегат f_Σ (рис.2.2).

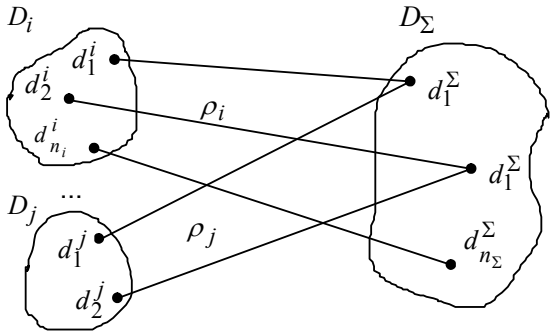


Рис. 2.2. Информационный межмодульный интерфейс

В традиционном программировании любое, даже незначительное изменение структур данных в модулях вызывает необходимость «ручной» переделки соответствующих информационных связей ρ_j. В технологии ГСП этот процесс автоматизирован за счет использо-

вания отношений ρ_j, описанных и хранящихся отдельно от программной реализации объекта в «паспортах» модулей.

Отношение ρ_j в ГСП формируется «паспортизацией» типов данных базовых модулей, т.е. за счет «опредмечивания» формальных параметров базовых модулей. В этом смысле отношение ρ_j является по сути «паспортом» модуля и вместе с базовым модулем определяют понятие актора или предиката (рис. 2.2).

2.5. Управление вычислительным процессом

В технологии ГСП (для объектов – агрегатов) в зависимости от стиля программирования (последовательного или параллельного) используются различные схемы управления вычислительным процессом. Каждому стилю программирования соответствует собственная схема управления в агрегате при незначительных изменениях в синтаксисе изображения агрегата.

Объекты, порожденные из базовых модулей (акторы), управляются в соответствии с правилами, принятыми в базовом языке программирования. Синтез агрегата происходит на основе его графического изображения, представленного соответствующими структурами данных, которые хранятся в информационном фонде технологии ГСП.

В ГСП агрегат фактически состоит из двух компонент:

- универсальной управляющей программы граф-машины (ГМ), которая в соответствии со структурой графа управления каждого из агрегатов управляет развитием вычислительного процесса на агрегате ПрОП;
- структур данных описания графа управления каждого из агрегатов.

Централизация функций управления в рамках одной программы (граф-машины) на самом деле очень удобное решение, поскольку позволяет:

- контролировать вычислительный процесс в целом. И в случае нештатных ситуаций принимать системные решения;
- реализовать сбор статистической информации о характеристиках надежности каждого из модулей, вычислительной сложности модулей, маршрутах развития вычислительного процесса и т.д.

Для описания структуры данных графа используется оригинальный способ представления графа с помощью так называемых структур смежности графа на смежной памяти. Введем две структуры, предназначенные для описания вершин графа и дуг переходов на графе:

```
typedef struct _ListTop
   {       char NameT[9];
         int  FirstDef;
          int  LastDef;
   } DEFTOP;
```

```
typedef struct _ListGraf
   {char NameA[9];
    int NambArc;
     int NambTop;
     int ArcType;
   } DEFGRAF;
DEFTOP    ListTop[6];
DEFGRAF  ListGraph[8];
```

В массиве ListTop каждой вершине массива ставится в соответствие участок массива ListGraph, на котором последовательно перечислены смежные с ней вершины графа. Начало и конец участка определяют параметры FirstDef и LastDef. В массиве ListGraph хранится описание дуг графа (NameA – имени дуги, ArcType – типа дуги) и номера вершины графа NambTop, смежной с родительской вершиной. При этом номер вершины совпадает с номером элемента массива ListTop, где приводится полное описание вершин графа. Такой способ представления графа позволяет организовать быструю навигацию по структуре ориентированного графа в граф-машине GM. Например, для графа, представленного на рис. 2.3, соответствующие массивы показаны на рис. 2.4. Кодом «-77» отмечена концевая вершина.

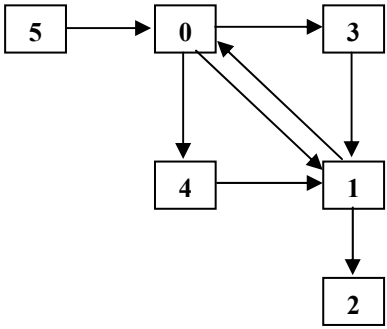


Рис. 2.3

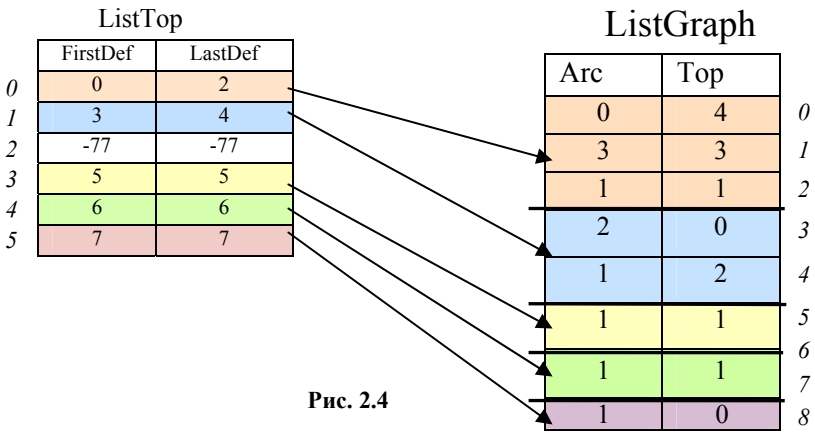


Рис. 2.4

Таким образом, содержательно агрегат состоит из декларирующей части, в которой описываются используемые в нем объекты вместе с графом управления создаваемого модуля, а также вызова граф-машины, управляющей вычислительным процессом (рис. 2.5). Вычислительный процесс развивается под управлением GM в соответствии с информацией, «заложенной» в управляющих структурах данных агрегата.

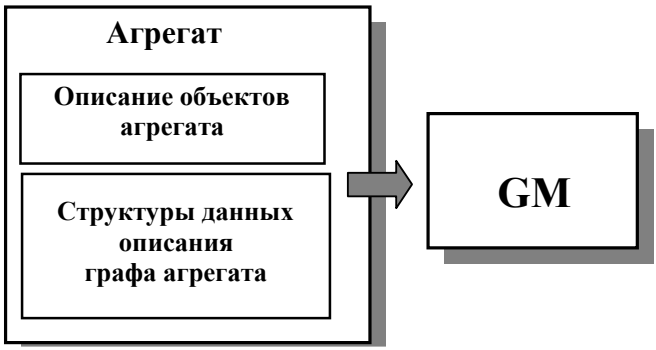


Рис. 2.5. Функциональная схема агрегата

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

Последовательное (процедурное) программирование в технологии ГСП реализуется механизмом, который определяется как механизм *интеррогативного управления*.

В основе такого управления лежит представление о том, что граф-машина GM способна обеспечить постоянную проверку условий перехода граф-программы из одного состояния в другое. Логика развития вычислительного процесса в этом случае определяется логикой изменения значений системы предикативных функций, разрешающих или запрещающих переходы программы из одного состояния в другое.

В каждый конкретный момент времени граф-машина GM находится в одной из вершин графа (в состоянии выполнения соответствующего объекта граф-программы), которую будем называть *текущей вершиной графа*.

Управляющая программа GM на каждом шаге работы обеспечивает переход из текущей вершины A_k в вершину инцидентную ей, например, как это показано на рис. 2.6 – в одну из вершин списка A_{i_0}, A_{i_1},.... A_{i_L}. Переход возможен, если соответствующий предикат $\mathrm{P}_{i_k}^{k}$ принял значение «истина» («1»). Во избежание неоднозначности при реализации перехода в случае, когда одновременно несколько предикатов приняли значение «1», дуги помечаются приоритетными индексами, причем чем меньше индекс, тем выше приоритет. Поэтому переход на графе будет реализован по дуге, для которой предикат принял значение «истина» и которая имеет наивысший приоритет. Для простоты положим, что верхний индекс в обозначении предиката устанавливает приоритет дуги.

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

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

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

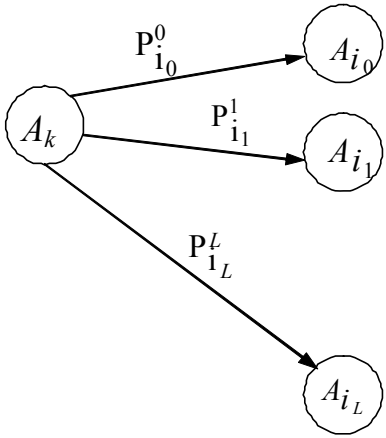


Рис. 2.6. Фрагмент граф-программы

2.6. Модель алгоритма решения задачи «Ханойские башни»

Задача о «Ханойских башнях» – одна из наиболее известных головоломок, для которой разработано большое количество алгоритмов её решения. Суть задачи заключается в следующем. Пусть имеются три стержня X, Y и Z. На стержень X надето N дисков разного диаметра, упорядоченные в направлении убывания диаметров от основания стержня. Цель игры заключается в переносе всех дисков со стержня X на стержень Z (по одному диску за раз), используя при этом промежуточный стержень Y, причем ни один диск большего диаметра нельзя ставить на диск меньшего диаметра.

Наиболее известным решением данной задачи, вошедшим во многие учебные пособия, является рекурсивный алгоритм [16]. Автоматный подход к решению задачи о «Ханойских башнях», позволивший устранить рекурсию, предложен в работе [20]. Однако все они так или иначе основаны на идеях рекурсивного алгоритма, т.е.

при составлении программ заранее был известен алгоритм решения задачи.

Попытаемся написать программу для задачи о «Ханойских башнях», не имея перед собой алгоритма её решения. Будем руководствоваться следующими простыми принципами, входящими в условия задачи:

1. При перекладывании дисков всегда применяется правило о недопустимости установки диска большего диаметра на диск меньшего диаметра.

2. Соблюдать правило приоритетов для операций переноса дисков. Установить приоритеты операциям переноса дисков (в порядке убывания приоритетов) в следующем порядке: $(X \to Z), (X \to Y), (Y \to Z), (Y \to X), (Z \to Y), (Z \to X)$.

3. В первую очередь снимать диски со стержня X и переносить их на стержень Z, если это возможно.

Сформируем словарь данных предметной области. Данные, необходимые для решения задачи, приведены в табл. 2.1. Числом 777 обозначено основание стержней.

Таблица 2.1. Словарь данных предметной области

Имя данного	Тип	Нач. значение	Комментарий
Mx	STERJN	{1,2,3,5,6,777}	Массив дисков стержня X
My	STERJN	{777,0,0,0,0,0}	Массив дисков стержня Y
Mz	STERJN	{777,0,0,0,0,0}	Массив дисков стержня Z
X	int	1	Размер верхнего стержня на диске X
Y	int	777	Размер верхнего диска на стержне Y
Z	int	777	Размер верхнего диска на стержне Z
Nst	int	1	Текущий номер стержня

В дальнейшем нам потребуется программа, выполняющая операцию переноса диска с одного стержня на другой.

Исходным «строительным» материалом для технологии ГСП являются базовые модули (функции интерпретации), составленные в кодах базового языка (в нашем случае С) и реализующие функциональные преобразования над данными ПрОП. Фактически они реализуют математические преобразования одних формальных параметров программного модуля в другие. Семантический смысл базовые мо-

дули приобретают после выполнения операции привязки формальных параметров модуля к данным предметной области, т.е. после выполнения операции паспортизации модуля.

В результате порождаются один или несколько акторов в технологии ГСП. Если базовые модули описывают абстрактные математические отношения над типами данных, то акторы выполняют вполне определенные преобразования данных предметной области и, с точки зрения её онтологии, формируют множество функций интерпретации *R*. На графе эти функции «привязываются» к его вершинам, образующим в совокупности множество понятий, входящих в онтологию ПрОП.

Рассмотрим базовый модуль *perA_B(int A, int B, STERJN Ma, STERJN Mb)*, реализующий перенос верхнего диска с абстрактного стержня A на абстрактный стержень B. Сама по себе программа базового модуля достаточно проста: первый элемент массива Ma переносится на первое место массива Mb (при этом реализуются все необходимые перемещения остальных элементов этих массивов) и переменным A и B присваиваются размеры первых элементов массивов Ma и Mb.

На основе базового модуля *perA_B* путём привязки формальных параметров A, B, Ma, Mb к данным ПрОП X, Y, Mx, My порождаются акторы, связанные с понятиями «Перенос диска со стержня X на стержень Y» (это понятие условно обозначим «X->Y»). Соответствующую функциональность реализует актор: *perA_B(X, Y, Mx,My)*. Точно также порождаются понятия «X->Z», «Y->Z», …., «Z->X» с помощью акторов: *perA_B(X, Z, Mx,Mz)*, *perA_B(Y, Z, My,Mz)*,…, *perA_B(Z, X, Mz,Mx)*.

Аналогичным образом вводятся понятие «Визуализация перемещений дисков» и актор, отображающий на экране дисплея положение дисков на стержнях: *move(Mx, My, Mz)*, а также понятия, связанные с управлением графического режима: «Инициализация графики», «Закрытие графического режима» и т.д.

Отметим, что многие понятия на схемах имеют свои графические образы, что упрощает восприятие граф-программ человеком.

Представим себе ситуацию, что уже выбран стержень, с которого будет сниматься диск. Необходимо построить алгоритм, определяющий, на какой стержень его необходимо переместить. Это несложно реализовать с помощью граф-программы «Перенос диска со стержня

X», представленной на рис. 2.7. В технологии ГСП такие объекты называются *агрегатами* и пополняют список множества функций интерпретации *R* онтологии предметной области.

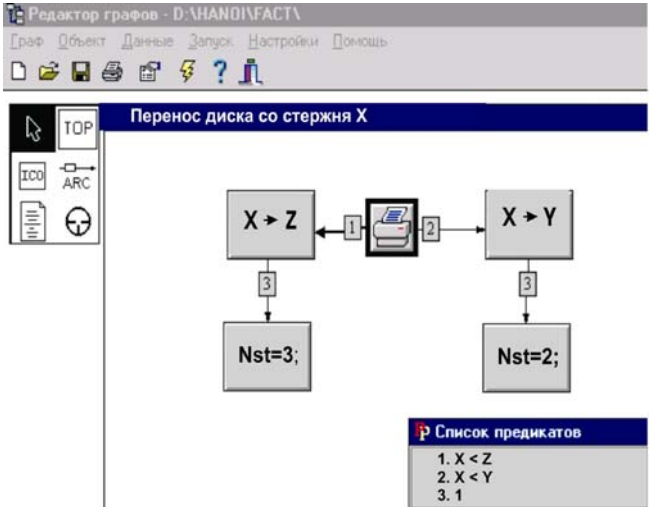


Рис. 2.7. Агрегат «Перенос диска со стержня X»

На рисунке корневая вершина (обведена жирной рамкой) выводит на экран дисплея текущее состояние стержней. Вершины «X->Z» и «X->Y» связаны с понятиями переноса диска со стержня X на стержни Z и Y. Причем переход в вершину «X->Z» происходит при выполнении условия X<Z, а в вершину «X->Y» – при истинности выражения X<Y. На графе более приоритетные дуги обозначены «жирными стрелками». Безусловный переход обозначен символом «1». Переменная *Nst* определяет номер стержня, на который реализован перенос диска. На графе этот факт отражен в вершинах «*Nst=3»;* и «*Nst=2»;* Стержням X, Y, Z присвоены номера 1, 2, 3.

Аналогично строятся агрегаты: «Перенос диска со стержня Y», «Перенос диска со стержня Z » (рис. 2.8).

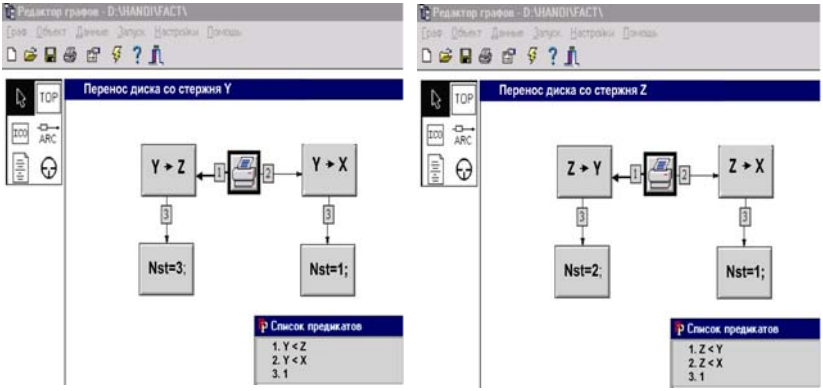


Рис. 2.8. Агрегаты «Перенос диска со стержня Y» и «Перенос диска со стержня Z»

Теперь построим основную граф-программу решения задачи о «Ханойских башнях».

Первоначально все диски находятся на стержне X. С диска X в общем случае мы можем переместить верхний диск либо на стержень Z, либо на стержень Y. Начальные действия построения агрегата «Ханойские башни» показаны на рис. 2.9. Здесь вершина *1* – «Инициализация графики» устанавливает графический режим отображения информации.

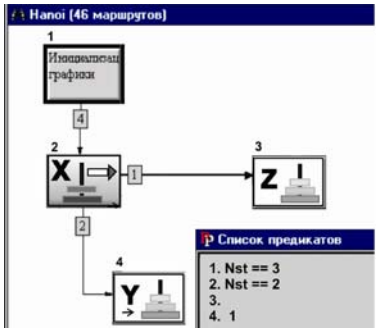


Рис. 2.9. Первый этап построения агрегата «Ханойские башни»

Вершина *2* привязана к агрегату «Перенос диска со стержня X», представленного на рис. 2.9. Вершины *3* и *4* формально фиксируют факт переноса диска на стержни Y или Z (управляются предикатами *Nst==3* и *Nst==2*). Эти вершины не имеют привязок к каким-либо акторам (т.е. являются «пустыми»), но необходимы для исключения циклического переноса дисков между двумя стержнями. Иными словами, если, например, диск был перенесен на стержень Y, то выбор стержня для выполнения следующей операции должен быть реализован между стержнями X и Y. Это обстоятельство приводит к развитию алгоритма решаемой задачи, представленной на рис. 2.10.

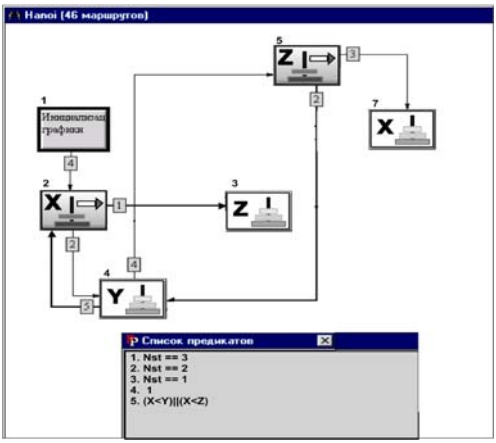


Рис. 2.10. Второй этап построения агрегата «Ханойские башни»

Как видно из рисунка вершина *4* связана с вершинами *5* и *2*, т.е. после переноса диска на стержень Y рассматриваются варианты возможности «снятия» дисков со стержней X или Z. В частности, переход в вершину *2* возможен в случае, если со стержня X имеется возможность переноса дисков на другие стержни, что управляется предикатом *5*: (X<Y)||(X<Z). Иначе следующий диск «снимается» со стержня Z.

Очевидно, что, если мы будем снимать диск со стержня *Z* (вершина *5*), то в качестве «операционных» стержней следует рассматри-

вать стержни X и Y (вершины *4* и *7*). Продолжая в том же духе, получим окончательный вариант агрегата «Ханойские башни», обеспечивающий решение поставленной задачи (рис 2.11).

Технологические акторы *9, 10* и *11* необходимы для отображения результата работы алгоритма и закрытия графического режима работы монитора. Предикат №7 обеспечивает остановку работы алгоритма.

Таким образом, работая над построением модели алгоритма решения задачи о «Ханойских башнях», мы фактически построили онтологию в этой предметной области, основываясь на правилах, принятых в рассматриваемой игре, и применяя разумные эвристики. Не менее важным обстоятельством является то, что на основе построенного алгоритма решения игровой задачи в технологии ГСП автоматически генерируется исполнимый код программы.

2.7. Краткий обзор раздела

В данной главе рассмотрены основные принципы построения технологии визуального программирования, получившей название *технологии графосимволического программирования* [4].

Визуальное программирование повышает наглядность представляемых кодов, существенно уменьшает число ошибок, допускаемых на этапе проектирования и кодирования программ, и тем самым повышает надежность кодов разрабатываемых программ.

Предложен оригинальный *метод организации межмодульного информационного интерфейса*, который позволяет описывать независимо друг от друга программные коды модулей и соответствующие информационные связи по данным.

Таким образом, в ГСП реализована более глубокая структуризация разрабатываемых программ, когда изображение алгоритма программы, описание логических условий, управляющих развитием вычислительного процесса, межмодульный интерфейс и универсальный метод управления ходом вычислительного процесса описываются и хранятся независимо друг от друга.

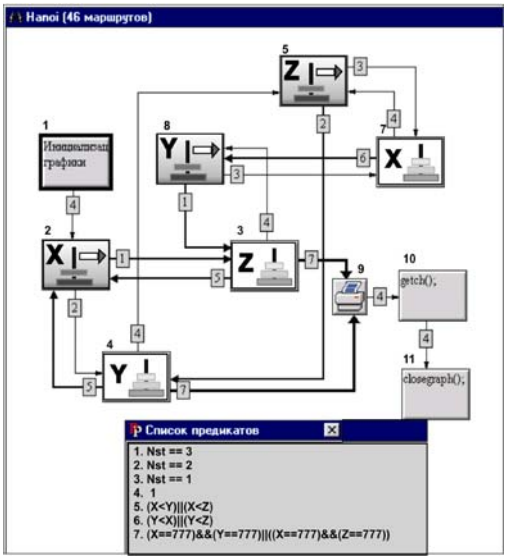


Рис. 2.11. Агрегат «Ханойские башни»

2.8. Контрольные вопросы

1. В чем состоят исходные положения технологии графосимволического программирования?
2. Какую алгоритмическую модель использует технология ГСП?
3. Какие объекты использует технология ГСП?
4. Опишите содержательно акторы, предикаты, агрегаты?
5. Что такое тип данного в ГСП?
6. Дайте определение типа базового модуля?
7. Какую роль в ГСП играют базовые модули?
8. Чем отличаются агрегаты от in-line модулей?
9. Каким образом в технологии ГСП организован межмодульный информационный интерфейс?
10. Как в технологии ГСП происходит управление вычислительным процессом? Для агрегатов. Для акторов.
11. Роль граф-машины в управлении агрегатов технологии ГСП.
12. Опишите способ представления графа управления, принятый при компиляции агрегата.

Глава 3. КОНСТРУИРОВАНИЕ ОБЪЕКТОВ ТЕХНОЛОГИИ ГСП

3.1. Введение

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

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

Однако автоматический синтез программ во всех областях применения компьютеров пока невозможен. Более того, он подходит лишь в достаточно простых и хорошо изученных областях, для которых легко определить и описать базу знаний. К сожалению, такая ситуация встречается нечасто.

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

Не случайно, что в настоящее время появилось большое количество визуальных средств программирования. К ним относятся разнообразные средства разработки оконных интерфейсов, «визарды» в языках управления базами данных, графические инструментальные

средства в CASE-системах, многочисленные проблемно или предметно-ориентированные языки программирования.

В то же время визуальное программирование расширяет «армию труда» в сфере программирования, поскольку к узкому кругу профессиональных программистов в новых условиях подключается большое количество «любителей», способных разрабатывать качественные программные продукты на новых средствах программирования.

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

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

Можно выделить три способа автоматического синтеза объектов ГСП (рис. 3.1): *паспортизацию, агрегацию и инкапсуляцию*.

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

Агрегация из совокупности имеющихся семантических конструкций порождает новую семантическую конструкцию.

Инкапсуляция из выбранной семантической формы объекта формирует новую синтаксическую конструкцию (базовый модуль). В определенном смысле эта операция выполняет действие, обратное двум предыдущим.

В данной главе рассматриваются вопросы конструирования объектов на концептуальной основе технологии графосимволического программирования, а также проблема классификации данных агрегатов ПрОП.

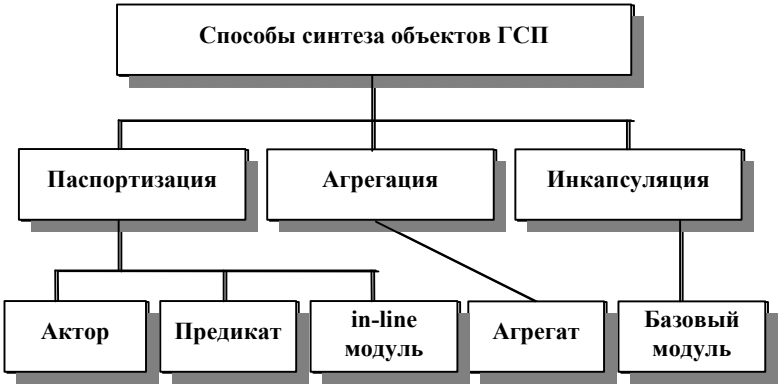


Рис. 3.1. Способы синтеза объектов ГСП

3.2. Конструирование объектов паспортизацией базовых модулей

Процесс порождения акторов из базовых модулей опишем на примере разработки граф-машины, используемой в технологии ГСП.

В табл. 3.1 для этого случая представлен словарь данных.

Таблица 3.1. - Словарь данных

Данные	Тип	Наименование	Начальное значение
Dop	voidU	Указатель на данные ПрОП	NULL
Fperd	int	Значение истинности предиката	0
J	int	Указатель на вершины, смежные с текущей	0
ListGraf	DEFGRAF	Описание структуры графа	{}
ListPred	DEFPRED	Список предикатов	{}
ListTop	DEFTOP	Список вершин графа	{}
Ntop	int	Число вершин графа	0
jPred	int	Номер предиката	0
jTop	int	Номер вершины	0
jroot	int	Номер корневой вершины	0
jfirst	int	Начало списка смежных вершин	0
jlast	int	Конец списка смежных вершин	0

Типы данных DEFGRAF и DEFTOP описаны в разделе 2.5. Типы данных DEFPRED и voidU имеют следующую нотацию:

```
typedef struct _ListPred
  {char NamePred[9];
   int (*Predicate)(void*); } DEFPRED;

typedef void* voidU[];
```

Тип DEFPRED[] описывает массив имен предикатов и имен соответствующих программных модулей.

Рассмотрим базовый модуль **GM_fl.c**, предназначенный для определения в ListGraf начала и конца подсписка вершин, смежных с текущей вершиной графа:

```
#include "stype.h"
int GM_fl(int *t1,int *t2,int *t3,DEFTOP *t4)
{
 *t1 = t4[*t3].FirstDef;
 *t2 = t4[*t3].LastDef;
return 1;
}
```

Порождение актора «Выделение списка смежных вершин для корневой вершины» реализуется путем построения паспорта данного объекта (рис.3.2).

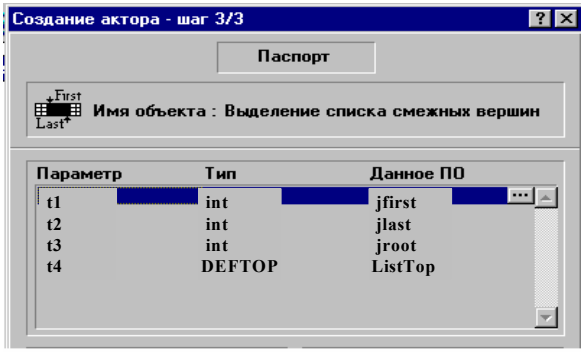


Рис. 3.2. Паспорт актора "Выделение списка смежных вершин для корневой вершины"

При этом в ПрОП автоматически порождается модуль:

```
int saaaaaaa(void *_G)
{return GM_fl((int)_G.first,(int)_G.last,(int)_G.root,(DEFTOP )_
.LisatTop); }
```

Аналогично на основе базового модуля **GM_fl**, формируется актор «Выделение для текущей вершины списка смежных вершин» (рис. 3.3).

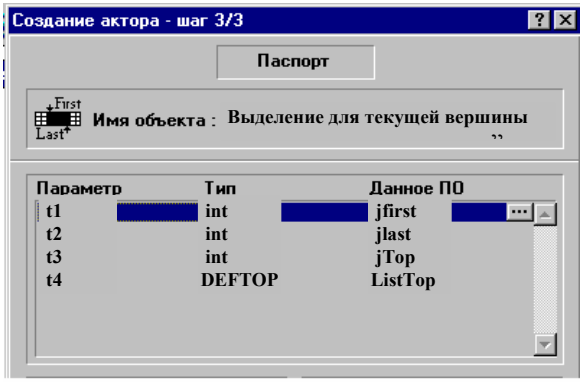


Рис. 3.3. Паспорт актора "Выделение для текущей вершины списка смежных вершин"

Полученные паспорта акторов хранятся в информационном фонде ПрОП.

Акторы типа «in line» порождаются автоматически из достаточно простых синтаксических конструкций базового языка, использующих имена данных ПрОП. Например, настройка указателя **J** на соответствующий элемент массива ListGraf[], представляемая в языке С++ выражением **J=jfirst**, после компиляции порождает стандартную для ГСП конструкцию:

```
#include "stype.h"
int aaaaaaaf(void *_G)
 { (*_G).J=(*_G).jfirst; return 1; }
```

При этом автоматически строится и паспорт модуля.

3.3. Конструирование агрегатов

В разделе 2.2 было указано, что агрегаты состоят из двух частей: автоматически компилируемого программного модуля, содержащего описание графа агрегата, и управляющей программы – граф-машины (GM). Принципы управления, используемые в GM, описаны в разделе 2.5. Алгоритм построения граф-машины представлен на рис. 3.4. Описание объектов агрегата GM приведено в табл. 3.2.

Таблица 3.2. Перечень объектов агрегата GM

Номер	Тип	Имя модуля	Назначение объекта
0	актор	saaaaaab	Вызов корневой вершины
1	актор	saaaaaaa	Выделение списка смежных вершин для корневой вершины
2	актор	aaaaaaaf	J=jfirst;
3	актор	aaaaaaah	jPred=ListGraf[J].NambPred;
4	актор	saaaaaac	Проверка предиката
5	актор	aaaaaaai	jTop=ListGraf[J].NambTop;
6	актор	saaaaaab	Вызов текущей вершины
7	актор	saaaaaaa	Выделение для текущей вершины списка смежных вершин
8	актор	aaaaaaaj	printf("ОШИБКА: Из вершины %s нет исходящей дуги\n",ListTop[jTop].NameTop);
9	актор	aaaaaaak	/* Конец */
10	актор	aaaaaaag	J++

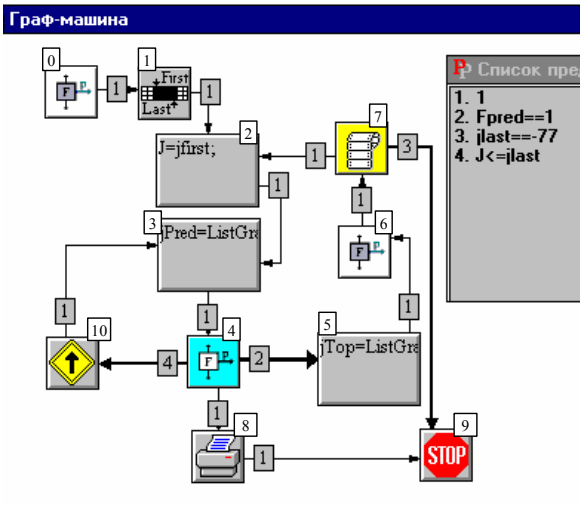


Рис. 3.4. Агрегат GM

Как видно из рисунка программы, граф-машина GM реализует циклический просмотр вершин, смежных с текущей, по маршрутам «M3-4-10-3» (если предикат, исходящий из текущей вершины, ложен) и «M2-3-4-5-6-7-2» (если предикат, исходящий из текущей вершины, принял значение – истина). Нормальное завершение процесса «блуждания» по графу алгоритма происходит, если из текущей вершины нет исходящих дуг (код вершины равен -77).

Компиляция «рисунка» агрегата порождает текст следующей программы:

```
#include "stype.h"
  int saaaaaab(void *);
  int saaaaaaa(void *);
  int aaaaaaaf(void *);
  int aaaaaaah(void *);
  int saaaaaac(void *);
  int aaaaaaai(void *);
  int aaaaaaaj(void *);
  int aaaaaaak(void *);
  int aaaaaaag(void *);
  int paaaaaaa(void *);
```

```
  int paaaaaab(void *);
  int paaaaaac(void *);
  int paaaaaad(void *);
  int GM(void *,int, int, DEFPRED*, DEFTOP*, DEFGRAF*);
static DEFPRED ListPred[4]={{"paaaaaaa", paaaaaaa},
                            {"paaaaaab", paaaaaab},
                            {"paaaaaac", paaaaaac},
                            {"paaaaaad", paaaaaad}};
static DEFTOP ListTop[11]={ {"aaaaaaaa",0, 0, &saaaaaab},
                            {"aaaaaaad",1, 1, &saaaaaaa},
                            {"aaaaaaaf",2, 2, &aaaaaaaf},
                            {"aaaaaaah",3, 3, &aaaaaaah},
                            {"aaaaaaac",4, 6, &saaaaaac},
                            {"aaaaaaai",7, 7, &aaaaaaai},
                            {"aaaaaaab",8, 8, &saaaaaab},
                            {"aaaaaaae",9, 10, &saaaaaaa},
                            {"aaaaaaaj",11, 11, &aaaaaaaj},
                            {"aaaaaaak",-77, -77, &aaaaaaak},
                            {"aaaaaaag",12, 12, &aaaaaaag}};
 static DEFGRAF ListGraf[13]={ {0, 1 },
                               {0, 2 },
                               {0, 3 },
                               {0, 4 },
                               {1, 5 },
                               {3, 10 },
                               {0, 8 },
                               {0, 6 },
                               {0, 7 },
                               {2, 9 },
                               {0, 2 },
                               {0, 9 },
                               {0, 3 }};
int gaaaaaaa(void *p[])
   { int Ntop =11;   int jroot = 0;
  GM(_G,jroot,Ntop,ListPred,ListTop,ListGraf);
  return(1);
}
```

Поскольку паспорт агрегата является композицией паспортов акторов и предикатов, включенных в агрегат, то паспорт агрегата не сохраняется в информационном фонде ГСП. Однако он легко строится автоматически.

3.4. Классификация данных объектов ГСП

3.4.1. Проблема классификации данных агрегатов

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

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

1) при построении на основе объекта исполняемого EXE-модуля;
2) при тестировании объектов технологии ГСП.

Потребность в классификации данных связана с необходимостью инициализации входных данных. Так, например, отсутствие начальных значений входных или модифицируемых переменных непременно вызовет ошибочную ситуацию при исполнении EXE-модуля.

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

Для объектов типа акторов и предикатов вопрос классификации данных решается пользователем на этапе формирования паспорта модуля. При классификации данных агрегата задача становится не столь очевидной. В отличие от актора в агрегате отнесение того или иного данного к определенному типу в значительной степени зависит от маршрута работы алгоритма на управляющем графе объекта.

Пусть мы имеем агрегат G_1, представленный на рис. 3.5. Предположим, что классификация параметров модулей A1, A2 и A3 представлена в табл. 3.3.

Таблица 3.3. Типы данных акторов

Модуль	Параметр	Тип
A1	d1	входной
A2	d2	вычисляемый
A3	d2	входной

Тогда, если алгоритм будет выполняться по маршруту

G1 :

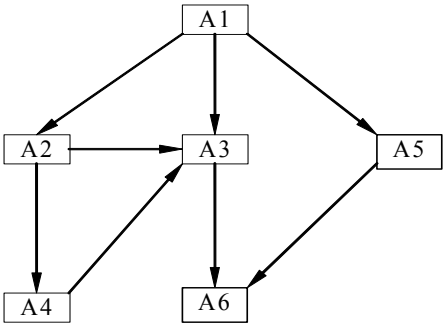


Рис. 3.5

A1→A3→A6, параметр d2 (из модуля A3) должен принять значение перед началом работы программы и его необходимо отнести к классу входных данных. Если же алгоритм пойдет по маршруту A1→A2→A3→A6, то d2 будет вычислен в модуле A2 и, следовательно, это данное не надо инициировать заранее (то есть оно является вычисляемым). Учитывая априорную неопределенность развития вычислительного процесса на графе G_1, данное d2 следует отнести к классу входных данных. Действительно, если вычисления «пойдут» по первому маршруту, то для него следует иметь начальное значение, если же вычисления будут развиваться по второму маршруту, то инициируемое значение будет заменено на вычисленное. В любом

случае не возникнет ошибочная ситуация, связанная с отсутствием или неверным заданием начального значения для данного d2.

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

3.4.2. Декомпозиция агрегатов. Алгебра трехзначной логики выделения классификационных признаков

Задача построения всех независимых маршрутов, исходящих из корневой вершины в концевые вершины графа, эквивалентна задаче декомпозиции исходного графа на совокупность частей графа, таких что $G = \bigcup R_j$, где R_j – ориентированные маршруты из корневой вершины в концевые.

Так, например, граф, представленный на рис. 3.5, может быть разложен на следующие маршруты (рис. 3.6):

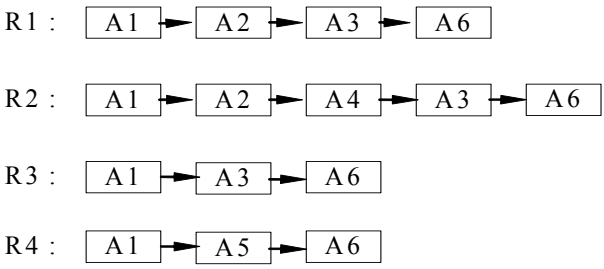


Рис. 3.6

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

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

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

Классификационные признаки данных для линейных графов определяются расположением объектов на маршруте вычислений. Если несколько модулей на маршруте имеют общий параметр, то его классификационный признак будет определяться типом его первого вхождения. Например, для маршрута R1: A1→A2→A3→··· параметр d2 встречается в двух модулях: A2 и A3, в модуле A2 этот параметр является вычисляемым и не требует предварительной инициализации, а в модуле A3 – инициируемым и должен принять значение заранее. Но так как модуль A2 в этой цепочке встречается раньше, то d2 не нужно заранее инициализировать. Таким образом, на этом маршруте параметр d2 необходимо отнести к классу *вычисляемых*. С другой стороны, если данное встречается впервые как инициируемое (внешнее), то его следует отнести к классу *внешних* данных.

Введем операцию конкретизации классификационных признаков для линейных маршрутов Δ. Применение этой операции к объектам маршрута порождают формулы вида

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

где $P(\ )$ – операция паспортизации соответствующего объекта.

После того как будут получены паспорта всех маршрутов, на следующем этапе классификации параметров агрегата необходимо объединить эти маршруты. При этом надо учитывать следующее обстоятельство: *если несколько маршрутов имеют общий параметр, то его классификационный признак будет определяться с учетом приоритетности признака необходимости инициализации данного*.

Предполагается, что признак инициируемости является более приоритетным по отношению к признаку вычислимости. Например, если параметр d2 на маршруте R1 определился как инициируемый (входной), а на маршруте R2 – как вычисляемый, то для нормальной работы агрегата этот параметр должен быть инициализирован заранее.

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

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

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

Определим семантику для введенных операций.

Пусть заданы два объекта ***A*** и ***B*** с соответствующими паспортами:

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

Обозначим классификационные признаки данных следующим образом:

0 – d $\notin$ P(A) — данное не принадлежит модулю A;

1 – d $\in D_A^{in}$ — данное принадлежит модулю A и является *входным* (инициируемым);

2 – d $\in D_A^{out}$ — данное принадлежит модулю A и является вычисляемым.

Тогда таблица истинности операций *конкретизации* Δ и *объединения* ∇ имеет вид (см. табл. 3.4):

Таблица 3.4

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

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

а) классификация данных агрегата при выполнении операции *конкретизации*:

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

где

$$D^{in}_{A\Delta B} = \left\{ d : d \in D^{in}_{A} \vee \left(d \notin P(A) \wedge d \in D^{in}_{B} \right) \right\}$$
$$D^{out}_{A\Delta B} = \left\{ d : d \in D^{out}_{A} \vee \left(d \notin P(A) \wedge d \in D^{out}_{B} \right) \right\};$$

б) классификация данных агрегата при выполнении операции объединения паспортов цепочек A и B (операция *объединения*):

$$P(A \nabla B) = \left\{ D^{in}_{A\nabla B}, D^{out}_{A\nabla B} \right\},$$

где

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

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

1. *Ассоциативность* операций конкретизации и объединения:

$$A \Delta (B \Delta C) = (A \Delta B) \Delta C = A \Delta B \Delta C = ABC,$$
$$A \nabla (B \nabla C) = (A \nabla B) \nabla C = A \nabla B \nabla C.$$

2. *Коммутативность* для операции объединения:

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

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

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

4. *Идемпотентность* операций конкретизации и объединения:

$$A \nabla A = A,$$
$$A \Delta B \Delta A = A \Delta B.$$

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

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

6. *Отбрасывание общего окончания* для операции конкретизации:

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

7. Свойство *симметричного поглощения* для операции конкретизации:

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

Представленные свойства операций конкретизации и объединения позволяют оптимизировать алгоритм классификации данных за счет сокращения количества маршрутов и уменьшения их длины. Рассмотрим возможный способ применения этих свойств на примере графа G1 (рис. 3.5).

На первом этапе для каждого из четырех маршрутов графа строятся паспорта, при этом используется операция конкретизации:

$$P(R1) = A1 \Delta A2 \Delta A3 \Delta A6,$$
$$P(R2) = A1 \Delta A2 \Delta A4 \Delta A3 \Delta A6,$$
$$P(R3) = A1 \Delta A3 \Delta A6,$$
$$P(R4) = A1 \Delta A5 \Delta A6.$$

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

$$P(G1) = P(R1) \nabla P(R2) \nabla P(R3) \nabla P(R4) =$$
$$= (A1 \Delta A2 \Delta A3 \Delta A6) \nabla (A1 \Delta A2 \Delta A4 \Delta A3 \Delta A6) \nabla \quad (3.1)$$
$$\nabla (A1 \Delta A3 \Delta A6) \nabla (A1 \Delta A5 \Delta A6).$$

Полученное выражение можно существенно сократить, если применить к нему преобразования, основанные на свойствах операций, в частности, свойство 6 – отбрасывание общего окончания для операции конкретизации.

Очевидно, что выражения для цепочек R1 и R2:

$$(A1 \Delta A2 \Delta A3 \Delta A6) \text{ и } (A1 \Delta A2 \Delta A4 \Delta A3 \Delta A6)$$

имеют общее начало (A1 Δ A2) и общее окончание (A3 Δ A6). На основании свойств 1 и 6 их можно преобразовать к виду:

$$(A1 \Delta A2 \Delta A3 \Delta A6)\nabla(A1 \Delta A2 \Delta A4 \Delta A3 \Delta A6) \nabla (A1 \Delta A3 \Delta A6) \nabla$$
$$\nabla (A1 \Delta A5 \Delta A6) = (A1 \Delta A2 \Delta A3 \Delta A6) \nabla (A1 \Delta A2 \Delta A4) \nabla$$
$$\nabla (A1 \Delta A3 \Delta \Delta A6) \nabla (A1 \Delta A5 \Delta A6).$$

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

$$P(G1) = (A1 \Delta A2 \Delta A4) \nabla (A1 \Delta A3 \Delta A6) \nabla (A1 \Delta A5 \Delta A6). \quad (3.2)$$

Последнее сокращает число операций классификации данных агрегата с 14 до 8 операций, т.е. на 43%.

3.4.3. Сжатие числа операций алгоритма классификации данных

К сожалению, количество потенциальных маршрутов графа комбинаторно растет с ростом цикломатического числа ν, характеризующего сложность программного модуля*. Из табл. 3.5 видно, что, начиная с ν=6, число потенциальных маршрутов полносвязного графа исчисляется десятками, а далее – сотнями тысяч. В этих условиях сравнительно простая процедура проверки принадлежности маршрута ориентированному графу становится невыполнимой, не говоря уже о классификации типов данных агрегата, когда число данных в каждом из модулей агрегата может исчисляться сотнями или тысячами.

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

Таблица 3.5

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

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

П1: $\dfrac{\alpha \Delta \beta \Delta \alpha}{\alpha \Delta \beta}$ – правило сокращения повторных вхождений,

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

* Цикломатическое число для графа вычисляется по формуле: $\nu = \gamma + m - n$, m – число ребер графа, n – число вершин графа, γ – число связных компонент графа

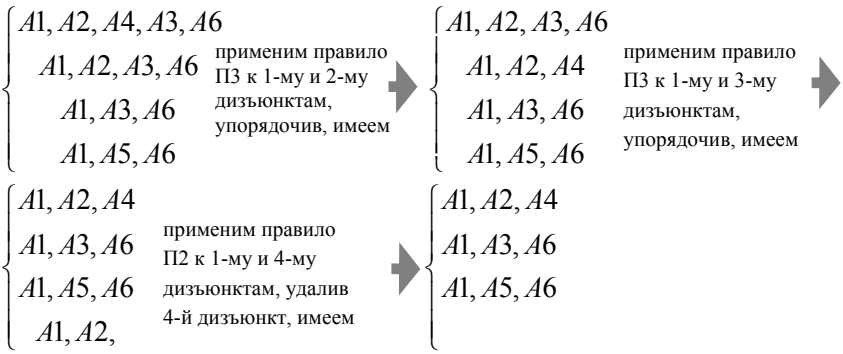
П3: $\dfrac{(\alpha\Delta\beta\Delta\gamma)\nabla(\alpha\Delta\gamma)}{(\alpha\Delta\beta)\nabla(\alpha\Delta\gamma)}$ поглощение общего окончания.

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

Алгоритм упрощения формулы классификации данных состоит из следующих шагов:

1. Все дизъюнкты упорядочиваются в первую очередь по длине и лексикографически в случае одинаковой длины. Более длинные дизъюнкты ставятся на первое место.
2. К каждому дизъюнкту применяется правило П1 до тех пор, пока это возможно.
3. К каждому дизъюнкту, начиная с первого, применяется правило П3. Причем, контактный «партнер» ищется на множестве дизъюнктов меньшей длины. Если правило П3 реализуется, то множество дизъюнктов переупорядочивается.
4. После выполнения пункта 3 алгоритма для каждого дизъюнкта меньшей длины ищется дизъюнкт, его поглощающий, т.е. применяется правило П2. Поглощаемый дизъюнкт вычеркивается из списка дизъюнктов.

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



В результате исходная формула (3.1) упростилась до вида (3.2), количество операций сократилось с 14 до 8. Вычислительные эксперименты показали, что сокращение числа операций на реальных объектах в среднем составляет 50%, причем с ростом цикломатического числа, когда число маршрутов нарастает, производительность алгоритма сжатия операций увеличивается.

3.5. Алгоритм классификации данных. Схема маршрута

Сложность алгоритма косвенно оценивается с помощью цикломатического числа графа управления. Цикломатика обеспечивает перечисление простых и сложных циклов в графе, не рассматривая составные циклы, когда одна или несколько дуг повторяются. На рис.3.7 показан граф, имеющий составной цикл R={a, b, c, a, b, d}.

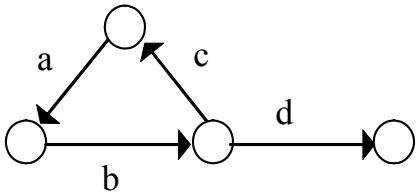


Рис. 3.7

Рассмотрим эвристический алгоритм классификации типов данных, основанный на реализации частичного перебора маршрутов граф-агрегата.

На этапе перечисления маршрутов из корневой вершины в концевые метод использует свойства алгебры 3-значной логики классификации данных.

Маршруты ориентированного графа будем описывать списками вершин. Данный подход менее удобен, чем при использовании списка дуг, поскольку часто содержит повторяющиеся вершины. Например, маршрут *M*={*A, B, E, F, B, C, D, B, G*} графа *G2* (рис. 3.8) повторяет вершину *B* три раза.

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

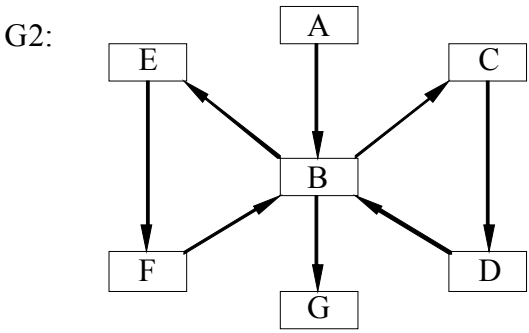


Рис. 3.8

ОПРЕДЕЛЕНИЕ. *Схемой маршрута* будем называть список вершин маршрута без повторений.

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

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

Алгоритм частичного перебора (АЧП) использует следующие правила:

1. Вершины графа кодируются символами упорядоченного множества любым способом, но так, чтобы корневая вершина имела наименьший код, а концевые – наибольшие значения.
2. На каждом шаге работы алгоритма для текущего состояния схемы маршрута S_i на орграфе ищется путь перехода из последней вершины S_i в любую свободную вершину схемы S_i.
3. Если переход *возможен*, то схема маршрута «расширяется» на один символ.

4. Если переход *невозможен*, то рассматривается следующая вершина списка свободных вершин, и так до тех пор, пока не кончится список вершин.
5. Если переход из текущей вершины в любую из свободных вершин (не содержащихся в схеме маршрута) *невозможен*, то происходит «откат» по схеме маршрута на один символ назад.
6. При достижении концевой вершины алгоритм «откатывает» на один символ назад.
7. Алгоритм завершает свою работу, если список свободных вершин корневой вершины исчерпан.

Рассмотрим наиболее важные свойства схем маршрутов.

Лемма 1. *Справедливо следующее:*
$(\forall \alpha \in N(A, B_{i_1}, A, B_{i_2}, ..., B_{i_\alpha}, A, ...)) \to (A, B_{i_1}, B_{i_2}, ..., B_{i_\alpha}, ...)$,

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

Доказательство

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

Лемма 2. $(\forall \alpha \in N(A, B_{i_1}, B_{i_2}, ..., B_{i_\alpha}, A, ...)) \to (A, B_{i_1}, B_{i_2}, ..., B_{i_\alpha}, ...)$,
т.е. повторное *вхождение вершины А в составной маршрут может быть сокращено до одного (первого) вхождения на схеме маршрута*.

Доказательство

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

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

Лемма 3. *Мощность множества составных маршрутов на конечном полносвязанном орграфе несчетна и равна* $2^{\aleph_0}$.

Доказательство

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

Утверждение 1. *Множество всех схем составных маршрутов полносвязанного орграфа конечно.*

Доказательство

Леммы 1 и 2 утверждают, что из любого бесконечного составного маршрута можно удалить все повторные вхождения вершин вплоть до первого вхождения. Но тогда схема маршрута содержит только неповторяющиеся вершины. Для конечного графа порядка n длина схемы маршрута не превышает n. Более того, общее число схем маршрутов на конечном алфавите не может быть более чем n! (числа перестановок символов вершин).

Схема маршрута $L = \{v_0, v_1, \ldots, v_m\}$ реализует сжимающее кодирование маршрутов на орграфах, поскольку описывает путь из корневой вершины в концевую $v_0 \to v_m$ через любое количество повторений, предшествующих v_m вершин. Схема маршрута позволя-

ет закодировать любые маршруты орграфа: простые, сложные и составные.

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

Покажем, что алгоритм АЧП реализует частичный перебор схем маршрутов. В качестве примера рассмотрим граф *G*1, вершины которого пронумеруем так, как это показано на рис. 3.9. Алгоритм АЧП формирует множество схем маршрутов, реализуя разбиение исходной задачи на взаимно независимые подзадачи в соответствии со стратегией И/ИЛИ-графов [10].

G1 :

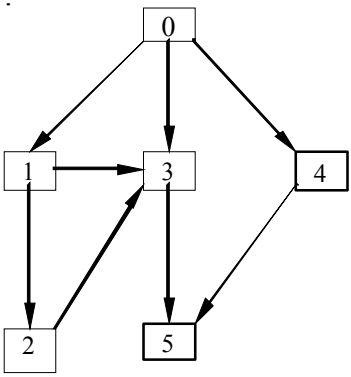


Рис. 3.9

Построим мультиграф с двумя типами вершин. Вершины типа «И» помечаются номерами вершин исходного графа, вершины типа «ИЛИ" помечаются списками вершин, достижимых (смежных) и недостижимых из соответствующей «И»-вершины графа. Правила алгоритма АЧП позволяют на каждом шаге алгоритма множество свободных вершин разбить на две части: *достижимое* и *недостижимое* подмножества. Причем достижимость понимается либо как непосредственный переход по дуге графа, либо опосредованно, через вершины предшествующего уровня иерархии, т.е. в соответствии со схемой маршрута.

На рис. 3.10, а показан И/ИЛИ-граф алгоритма АЧП для графа *G*1. Решением задачи перечисления схем маршрутов является подграф И/ИЛИ-графа, из которого исключены ИЛИ вершины (рис. 3.10, б).

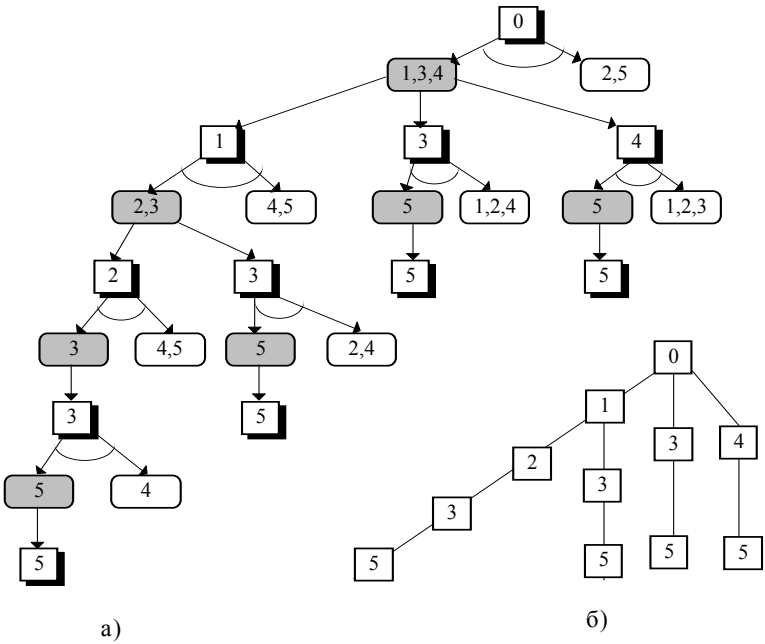


Рис. 3.10. И/ИЛИ-граф алгоритма АЧП

3.6. Эффективность алгоритма АЧП

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

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

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

Для того чтобы выяснить, как связана топологическая сложность графа со сложностью алгоритма, рассмотрим наиболее распространенный метод оценки топологической сложности, основанный на цикломатической метрике Маккейба [15].

В основу этого метода положена оценка сложности программы по числу линейно независимых маршрутов в его управляющем графе, то есть маршрутов, комбинируя которые, можно получить все пути из корневой вершины графа в концевую. Цикломатическая метрика программы определяется из ее управляющего графа следующим образом:

$$v_G = m - n + 1,$$

где m — число дуг в графе, n — число вершин.

Многие авторы подчеркивают адекватность этой метрики интуитивному пониманию сложности программного модуля. На основании опыта программирования и статистического материала [9] была определена разумная верхняя граница сложности v_G, равная 10. Рассмотрим, как связана эта метрика со сложностью алгоритма построения непериодических маршрутов, определяемой с помощью топологического дерева. С этой целью проведем эксперимент, который заключается в построении полного множества графов заданного порядка и оценки их сложности с помощью метрики Маккейба.

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

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

Зависимость разброса усредненной сложности алгоритма АПЧ в зависимости от порядка графа получена экспериментально, в результате моделирования орграфов различной структуры (рис. 3.11). Исследовались различные топологические варианты орграфов: от линейных графов (нижняя линия), до полносвязанных графов (верхняя линия). «Средняя» линия на рисунке разбивает приблизительно пополам множество топологически разных графов. Около этой линии «концентрируется» наибольшее число топологически различающихся орграфов одинакового порядка. Из рисунка видно, что с ростом ***n*** сложность алгоритма АЧП увеличивается даже для линейных графов и рациональным порядком графа можно считать $n \in [6,8]$. Примерно этого же уровня сложности следует придерживаться при разработке граф-агрегатов.

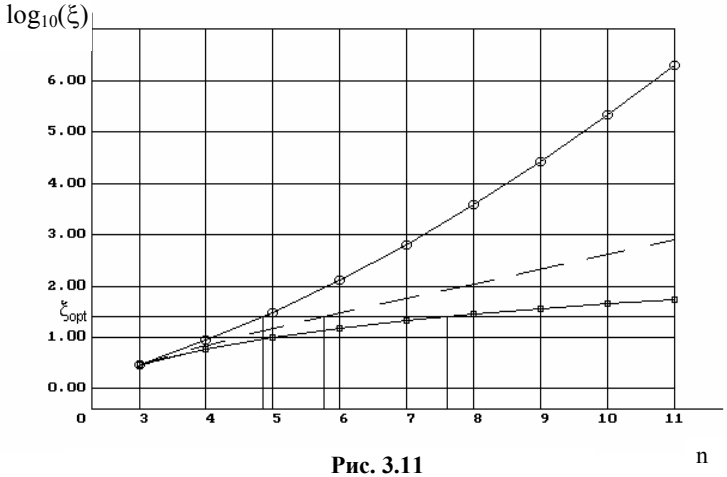


Рис. 3.11

3.7. Краткий обзор раздела

В главе рассматривались способы конструирования основных объектов (программных модулей) технологии ГСП. Технология графосимволического программирования предоставляет пользователю удобный графический язык для описания алгоритмов программ, позволяющий сократить сроки разработки и модификации ПО, повы-

сить наглядность представления программ, реализовать более надежный стиль программирования.

Подробно рассмотрена проблема классификации данных программных модулей. Выделены три типа основных данных: исходные, модифицируемые и вычисляемые. Показано, что при разработке сложного программного обеспечения, содержащего большое количество переменных, задача классификации данных по их использованию не столь тривиальна. В то время как неинициализированные исходные данные порождают большое количество непредсказуемых ошибок при исполнении кодов ПО. В режиме автоматического порождения кодов граф-программ проблема классификации данных вообще превратилась бы в неразрешимую проблему, если бы она в ГСП не решалась автоматически.

Предложенный в главе алгоритм классификации данных основан на алгебраическом подходе, использующем алгебру трехзначной логики. Более того, предложенная алгебра позволила ввести понятие схемы маршрута, полезное не только для классификации данных, но для структурного анализа разрабатываемых алгоритмов. В общем случае любой алгоритм, имеющий конечный граф управления, имеет бесчисленное множество маршрутов, что существенно осложняет тестирование широкого класса часто используемых алгоритмов. В то же время, как показано в главе, число схем маршрутов всегда конечно. И как результат – конечно и число тестов, которые необходимо реализовать в процессе исследования разрабатываемых программ.

В завершении главы приводится эффективный алгоритм частичного перебора схем маршрутов граф-программы. Исследуется эффективность этого алгоритма. Показано, что для обеспечения эффективного тестирования агрегатов в приемлемые сроки порядок агрегата в среднем не должен превышать 6...8 вершин графа.

3.8. Контрольные вопросы

1. Перечислите способы синтеза объектов технологии ГСП.
2. Как порождаются акторы и предикаты в технологии ГСП? Что такое «паспортизация» модулей?
3. Как строятся агрегаты?

4. Объясните алгоритм работы граф-машины, роль граф-машины в технологии ГСП.

5. Основные компоненты автоматически порождаемой граф-программы.

6. В чем состоит проблема классификации данных агрегатов?

7. Типы данных. Выделение маршрутов.

8. Дайте определение операциям объединения и конкретизации.

9. Свойства операций объединения и конкретизации.

10. Опишите алгоритм классификации данных.

11. Дайте определение понятию схемы маршрута. Чем оно отличается от классического понятия маршрута?

12. Опишите алгоритм сжатия числа операций классификации данных. Правила вывода алгоритма «упрощения» формул классификации данных.

13. Какие вершины графа управления называют *свободными вершинами* графа G относительно схемы маршрута S?

14. Опишите алгоритм частичного перебора классификации данных.

15. Эффективность алгоритма частичного перебора.

ГЛАВА 4. МОДЕЛИРОВАНИЕ ПАРАЛЛЕЛЬНЫХ ВЫЧИСЛЕНИЙ

Преимущества графических моделей раскрываются в полной мере при описании параллельных вычислений.

Текстовая форма представления информации по своей сути является последовательной. Текст состоит из последовательности предложений. Предложение формируется последовательностью слов. Каждое слово – это последовательность букв.

Для описания следующих друг за другом команд последовательного алгоритма такая форма представления является естественной. Однако она становится неудобной, если требуется описать параллельные взаимодействия. В этом случае разработчик вынужден мысленно строить модель параллельной программы на основе её последовательного текстового описания и абстрактно представлять себе, как будут взаимодействовать ее параллельные участки. Для этого разработчик должен разбираться в особенностях технологии распараллеливания, используемой в программе. Он не только должен понимать основные принципы этой технологии, но и детально знать синтаксис и семантику текстовых конструкций, используемых для распараллеливания. Фактически от разработчика требуется квалификация программиста, что препятствует широкому применению параллельных вычислений специалистами в различных предметных областях.

Графическая форма записи алгоритма позволяет явным образом изобразить параллельные фрагменты. Она более выразительна и компактна, чем текстовая. При графическом описании алгоритма разработчик имеет возможность визуализировать свою мысленную модель параллельного взаимодействия. Наличие визуальной модели облегчает также дальнейшую модификацию программы, поскольку она лучше запоминается и легче воспринимается, чем текст.

Для описания параллельных алгоритмов в технологии ГСП ее графическая модель дополняется новыми элементами. Семантика описанных выше базовых элементов модели остается прежней, что позволяет строить параллельную программу на основе уже созданной «последовательной» граф-модели. Тем самым облегчается переход к параллельному программированию. Ниже приводится подробное

описание процесса разработки параллельных программ в технологии ГСП.

4.1 Концептуальная модель организации параллельных вычислений в технологии ГСП

В параллельной программе, в отличие от последовательной, одновременно существует несколько вычислительных процессов. Вычисления в каждом из процессов выполняются последовательно, поэтому процесс может быть описан обычной «последовательной» граф-моделью. Для того чтобы в рамках одной модели изобразить несколько таких процессов, ее необходимо расширить. Прежде всего, необходимо выделить различные процессы, обозначить их начало и конец, определить условия их запуска и завершения. Это достигается расширением понятия «дуга» за счет введения дуг различного типа.

Определим формально тип дуги Ψij как функцию $T(\Psi ij) \in \{1,2,3\}$, значения которой имеют следующую семантику:

$T(\Psi ij) = 1$ – последовательная дуга (описывает передачу управления на последовательных участках вычислительного процесса);

$T(\Psi ij) = 2$ – параллельная дуга (обозначает порождение нового параллельного вычислительного процесса);

$T(\Psi ij) = 3$ – терминирующая дуга (завершает параллельный вычислительный процесс).

Для описания отдельного вычислительного процесса вводится понятие параллельной ветви β – подграфа графа G, начинающегося параллельной дугой (тип этой дуги $T(\Psi ij) = 2$) и заканчивающегося терминирующей дугой (тип этой дуги $T(\Psi ij) = 3$). $\beta = <A_\beta, \Psi_\beta, R_\beta>$, где A_β – множество вершин ветви, Ψ_β – множество дуг управления ветви, R_β – отношение над множествами вершин и дуг ветви, определяющее способ их связи.

Дуги, исходящие из вершин параллельной ветви β, принадлежат также ветви β. При кодировании алгоритма, описанного с помощью предлагаемой модели, каждая параллельная ветвь порождает отдельный процесс – совокупность подпрограмм, исполняемых последовательно на одном из процессоров параллельной вычислительной системы.

Графическая модель обычно содержит несколько параллельных ветвей, каждая из которых образует отдельный процесс. В этом смысле модель параллельных вычислений можно представить как объединение нескольких параллельных ветвей:

$$G = \bigcup \beta_k \ ,$$

где β_k – параллельные ветви графа *G*.

Таким образом, распараллеливание вычислений возможно только на уровне граф-модели. Вычисления в пределах любого актора выполняются последовательно.

Число параллельных ветвей в модели фиксируется при ее построении, при этом максимальное количество ветвей не ограничивается. Каждая ветвь имеет ровно один вход и один выход, для обозначения которых в граф-модели используются два типа дуг: параллельная дуга и терминирующая дуга (рис. 4.1).

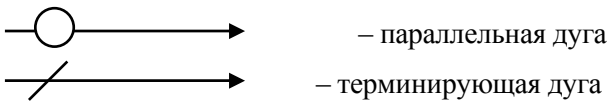


Рис. 4.1. Обозначение параллельных и терминирующих дуг в графической модели параллельных вычислений

Для описания правил построения граф-модели введем следующую систему обозначений:

To(*Ai*) – список дуг, входящих в вершину *Ai,*

Fr(Ai) – список дуг, исходящих из вершины *Ai*,

H(βj) – начальная вершина ветви *βj*,

| L | – число элементов в списке L.

Переход по параллельной дуге начинает работу параллельной ветви, переход по терминирующей дуге – заканчивает ее работу. Параллельная дуга не содержит предиката, т. е. переход по ней

$$\forall \Psi_{ij} \in \Psi, \quad T(\Psi_{ij}) = 2; \quad P_{ij} \equiv 1.$$

Из вершины может исходить не менее двух параллельных дуг. Если из вершины исходит параллельная дуга, то дуги других типов (обычная или терминирующая) не могут исходить из данной вершины:

$$T(\Psi_{i,j0}) = 2 \rightarrow \forall j \neq j0; \quad T(\Psi_{i,j}) = 2, \left|\mathrm{Fr}(A_i)\right| \geq 2.$$

Входящие в вершину дуги не могут быть одновременно параллельными и терминирующими:

$$T(\Psi_{i0,j0}) = 2 \rightarrow \forall i\, T(\Psi_{i,j0}) \neq 3.$$

Функционирование модели начинается с запуска единственной ветви, называемой мастер-ветвью (мастер-процессом). Обозначим мастер – ветвь $\beta 0$. В вершинах мастер-ветви, имеющих исходящие параллельные дуги, порождаются новые параллельные ветви. Вершины этих ветвей также могут иметь параллельные дуги, таким образом, допускается вложенность параллельных ветвей. Ветви, породившиеся в одной вершине некоторой ветви, должны терминироваться также в одной вершине этой же ветви:

$$\forall k\ H(\beta_k) = A_{k0}, \Psi_{i,k0} \in \mathrm{Fr}(A_i), A_i \in \beta_1 \rightarrow$$
$$\rightarrow \Psi_{kN} \in \mathrm{To}(A_j), T(\Psi_{kN}) = 3, A_j \in \beta_1.$$

В ветви β_l управление из вершины Ai после запуска ветвей β_k, $k = k1, k2, \ldots, K$ передается вершине Aj. Вершина Aj запускается на выполнение после завершения работы ветвей $\beta_{k1}, \ldots, \beta_K$. Таким образом, в каждый момент времени в любой ветви выполняется ровно одна вершина.

Переход между двумя вершинами, принадлежащими различным параллельным ветвям, возможен только по параллельным и терминирующим дугам, то есть запрещены условные переходы между вершинами различных параллельных ветвей:

$$\forall i, j,\ A_i \in \beta_l, A_j \in \beta_2 \rightarrow \mathrm{T}(\Psi_{ij}) \neq 1.$$

Если некоторая ветвь породила новые параллельные ветви, то вычисления в ней приостанавливаются до завершения работы порожденных ветвей. Таким образом, вложенные параллельные ветви выполняются последовательно относительно друг друга.

4.2 Модель синхронизации параллельных процессов

При создании математических моделей параллельных вычислений ключевой проблемой является синхронизация вычислений, т.е. организация их согласованного выполнения. Существуют различные способы синхронизации, такие как критические секции, семафоры, обмен сообщениями и мониторы [2]. В предлагаемой модели применяется комбинированный способ, использующий одновременно механизмы передачи сообщений и принципы мониторной синхронизации.

Определим почтовый ящик *Lpost* как список сообщений, с помощью которых вершины модели информируют друг друга о своем состоянии: $L_{post} = [\mu_{i_0,j_0},...,\mu_{i_m,j_n}]$, где $\mu_{i,j}$ – сообщение, посылаемое от вершины *Ai* вершине *Aj*. Возможность передачи сообщения от одной вершины граф-модели другой вершине изображается графически дугой синхронизации $\Psi_{i,j}$, проведенной из вершины-источника сообщения в вершину-получатель сообщения.

Вершина *Ak* посылает сообщения другим вершинам, записывая сообщения $L_{A_k Com} = [\mu_{k,j_0},...,\mu_{k,j_n}]$ в список *Lpost*. Наличие сообщения $\mu_{i,j}$ в списке *Lpost* говорит о том, что оператор вершины *Ai* завершил работу и хочет сообщить об этом вершине *Aj*.

Каждой вершине ставится в соответствие семафорный предикат $R_{Aj} = r(b_{i_0,j},...,b_{i_m,j})$, представляющий собой правильно построенную формулу относительно булевых переменных $b_{i_k,j}$, связанных логическими связками типа &, V. Булевы переменные определяются следующим образом:

$$b_{k,j} = \begin{cases} 1, & \text{если } \mu_{k,j} \in L_{post}, \\ 0, & \text{если } \mu_{k,j} \notin L_{post}. \end{cases}$$

Семафорный предикат разрешает или запрещает запуск соответствующей вершины. Если семафорный предикат $R_{Aj} = 1$, то запуск оператора вершины *Aj* разрешается. В противном случае вычисления приостанавливаются до того момента, когда R_{Aj} станет равным 1. Семафорный предикат определяется для всех вершин, в которые

входят дуги синхронизации. Для остальных вершин значение семафорного предиката принимается тождественно истинным.

Обмен сообщениями между вершинами, принадлежащими одной параллельной ветви, запрещен:

$$A_i \in \beta_k, A_j \in \beta_k \rightarrow b_{ij} \equiv 0.$$

Вершины в пределах одной параллельной ветви исполняются строго последовательно, поэтому их синхронизировать не требуется.

Замечание 4.1. Операцию "v" в семафорных предикатах следует использовать только над операндами, определенными над взаимоисключающими вершинами, то есть такими вершинами, факт запуска одной из которых исключает получение управления остальными вершинами. На рис. 4.2, а вершины А4 и А5 являются взаимоисключающими.

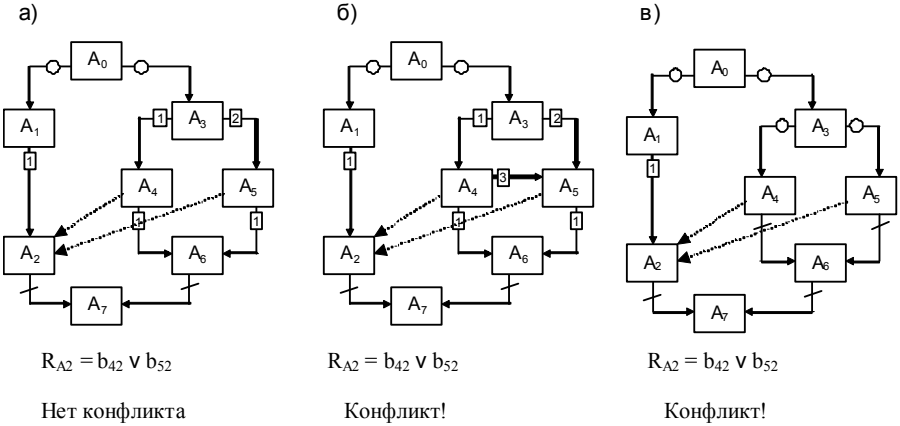


Рис. 4.2. Пример использования операции "v" в семафорных предикатах

Если вершины А2, А4 и А5 используют общее данное, а операторы в вершинах А2, А3 и А6 имеют относительно большую длительность исполнения, то имеет смысл ввести две дуги синхронизации $\Psi_{4,2}$ и $\Psi_{5,2}$ и определить семафорный предикат для вершины А2 с использованием операции "v": $R_{A_2} = b_{4,2} \vee b_{5,2}$. В случае передачи управления от вершины А3 к одной из вершин А4 или А5 (например, вершине А4) другая вершина (А5) уже не запустится, а значит, вершину А2 достаточно синхронизировать только с вершиной, полу-

чившей управление (А4). В этом случае для запуска на исполнение операнда в вершине А2 достаточно истинности одного из операндов ее семафорного предиката, и можно использовать операцию "v".

Если вершины не являются взаимоисключающими, то использование операции "v" над их сообщениями не разрешает конфликта совместного использования данных, так как приход сообщения от одной из таких вершин не исключает запуска другой на исполнение. Примеры подобных ситуаций приведены на рис. 4.2, б, в.

На рис. 4.2, б граф-модель дополнена дугой $\Psi_{4,5}$. Если управление передается по этой дуге от вершины А4 к вершине А5, то оператор последней может исполняться одновременно с оператором вершины А2, что приведет к конфликту совместного использования данных.

На рис. 4.2, в вершины А4 и А5 принадлежат различным параллельным ветвям. В этом случае использование операции "v" в семафорном предикате вершины А2 также не исключает конфликта, поскольку операторы вершин А4 и А5 могут исполняться одновременно и приход сообщения от одной из них не гарантирует, что оператор другой вершины также завершил исполнение.

Приведенные рассуждения и примеры показывают, что операцию "v" в семафорных предикатах следует применять с осторожностью, и только в тех случаях, когда легко видеть, что вершины, от которых исходят дуги синхронизации, являются взаимоисключающими.

4.3 Граф-машина для параллельных вычислений

В технологии ГСП используется централизованное управление процессом вычислений, осуществляемое специальной программой – граф-машиной. Рассмотрим реализацию граф-машины для параллельных вычислений.

Граф-машина универсальна для любого алгоритма. Анализируя его графическую модель, она выполняет в соответствующем порядке акторы и агрегаты, вычисляет значения предикатов и управляет синхронизацией. Для каждой параллельной ветви запускается по одному экземпляру граф-машины, которая представляет собой отдельный процесс в вычислительной системе. В граф-машинах различных па-

раллельных ветвей происходит прием и отправка сообщений и вычисление семафорных предикатов.

Опишем формально функционирование модели с помощью языка Р-схем [11].

Введем следующие обозначения:

- q – номер текущей вершины;
- GM(A_i) – запуск на выполнение граф-машины агрегата Ai;
- IncP(A_i) – операция построения списка дуг, инцидентных вершине Ai;
- Con(R_{Ai}) – список сообщений, контролируемых семафорным предикатом R_{Ai};
- Del[L1, L2] – операция удаления элементов списка L1 из списка L2.

Граф-машина начинает вычисления с запуска корневой вершины A_0. Далее обрабатываются все вершины граф-модели до тех пор, пока не будет достигнута концевая вершина или не возникнет ошибочная ситуация (рис. 4.3).

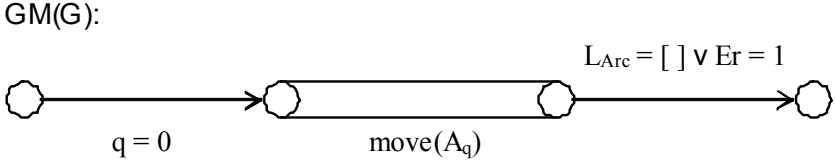


Рис. 4.3. Внешний цикл работы граф-машины

Концевая вершина не содержит исходящих дуг (IncP(A_i) = []). Ошибочная ситуация возникает в том случае, когда все предикаты, помечающие исходящие из некоторой вершины дуги, ложны. В этом случае происходит аварийное завершение процесса вычислений и граф-машина сигнализирует об ошибке.

Обработка очередной вершины выполняется по алгоритму move(A_q), изображенному на рис. 4.4.

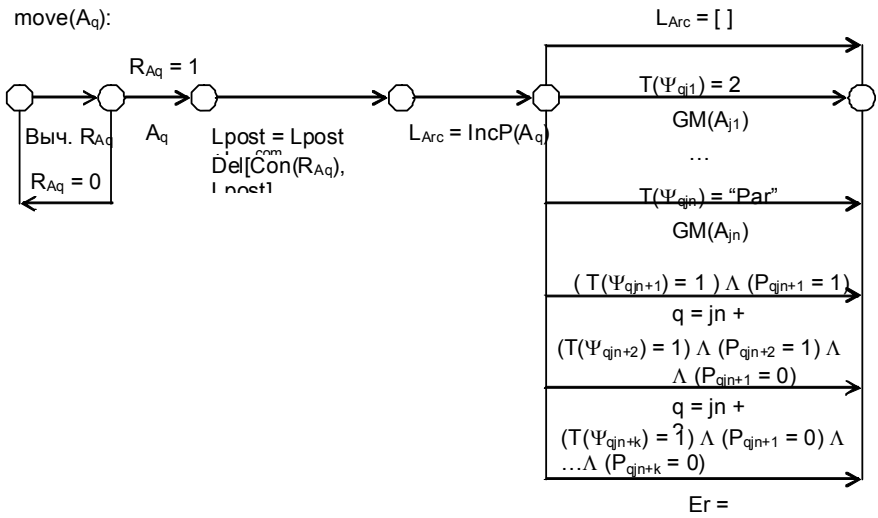


Рис. 4.4. Обработка вершины A_q граф-машиной

Выполнение актора, принадлежащего данной вершине, разрешается лишь в случае истинности ее семафорного предиката ($R_{Aq} = 1$). После выполнения актора в почтовый ящик отправляются сообщения, генерируемые текущей вершиной, и удаляются сообщения, контролируемые ее семафорным предикатом.

Затем строится список дуг, исходящих из текущей вершины. Этот список просматривается граф-машиной последовательно, начиная с самой приоритетной дуги. Вычисляется значение предиката, помечающего дугу, и в случае его истинности, происходит переход к обработке следующей вершины. В результате обработки параллельной дуги в отдельном процессе запускается другая граф-машина, обрабатывающая порождаемую данной дугой параллельную ветвь. После запуска всех параллельных ветвей происходит переход в вершину, в которой они терминируются. Запуск актора этой вершины контролируется ее семафорным предикатом. Если значение семафорного предиката не задано явно, он строится автоматически в виде конъюнкции сообщений от концевых вершин параллельных ветвей:

$$(R_{A_q} = b_{i1,q}) \wedge (R_{A_q} = b_{i2,q}) \wedge \ldots \wedge (R_{A_q} = b_{in,q})$$

Если семафорный предикат содержит дизъюнкцию некоторых сообщений, то в случае его истинности граф-машина принудительно завершает работу ветвей, сообщения от которых еще не пришли.

4.4 Реализация модели общей памяти технологии ГСП в распределенных системах с использованием технологии MPI

Технология ГСП определяет общий для всех модулей программы словарь данных. При выполнении созданной в технологии программы на параллельной вычислительной системе логично использовать модель общей памяти. Вместе с тем, большинство высокопроизводительных вычислительных систем в настоящее время являются кластерными системами, в которых каждый узел имеет локальную область памяти. Наиболее распространенный способ разработки программ для кластерных систем – использование технологии MPI. В этой технологии каждый процесс параллельной программы имеет собственное адресное пространство, а обмен данными между процессами осуществляется с помощью пересылки сообщений. Параллельный вариант граф-модели при использовании технологии MPI предполагает распределенное размещение данных на различных узлах вычислительной системы. Для соблюдения вышеназванных условий в параллельной версии технологии ГСП программно реализована модель общей памяти для аппаратной архитектуры с распределенной памятью.

4.4.1 Диспетчер данных

В среде выполнения программы выбирается вычислительный узел, на котором будет запущен процесс, отвечающий за хранение переменных ПрОП. Учитывая аппаратные особенности и топологию ВС, это может быть узел с наибольшим объемом оперативной памяти или центральный узел, имеющий минимальное время доступа от любого из остальных узлов кластера. Преимущество данного подхода в том, что значительно экономится ресурс памяти на вычислительных узлах, т.к. на узлах память выделяется только под те переменные, которые используются процессом, исполняющимся на данном узле.

Предлагаемый способ обмена данными требует введения понятия диспетчера данных – подпрограммы, выполняющей функции хранения, чтения и модификации данных предметной области.

Диспетчер данных, называемый также менеджером памяти, исполняется в отдельном процессе параллельной программы. Он порождает объект, описываемый классом TPOData, который хранит значения данных предметной области. В каждом из процессов, содержащих параллельные ветви граф-модели, также порождается объект класса TPOData. Однако функции доступа к членам-данным у объекта диспетчера данных и у объектов параллельных ветвей различаются. Диспетчер данных хранит все данные в локальной памяти и для обращения к ним использует обычные указатели. На остальных процессах используется ленивая инициализация памяти под переменную при первом доступе.

В параллельных ветвях граф-модели для чтения или записи некоторого данного осуществляется обращение к диспетчеру памяти с помощью совокупности сообщений. В первом сообщении пересылается запрос на чтение или запись конкретного данного. Каждая переменная из ПОП получает уникальный номер, по которому диспетчер памяти может ее идентифицировать. В случае чтения параллельная ветвь переходит к ожиданию ответа от диспетчера данных. При записи во втором сообщении пересылается новое значение данного. Диспетчер данных циклически принимает и обрабатывает запросы параллельных ветвей (рис. 4.5).

4.4.2 Обзор класса TPOData

Класс TPOData – это ядро механизма хранения и передачи данных в технологии ГСП. Класс TPOData инкапсулирует все данные ПОП и предоставляет доступ к ним через поля-свойства. Полей ровно столько, сколько переменных в ПОП, а их названия совпадают с названиями переменных. Для переменных простых типов в классе TPOData описано по одному методу доступа для чтения и установки. Для массивов определено по два метода чтения и установки: доступ ко всему массиву и к элементу по индексу.

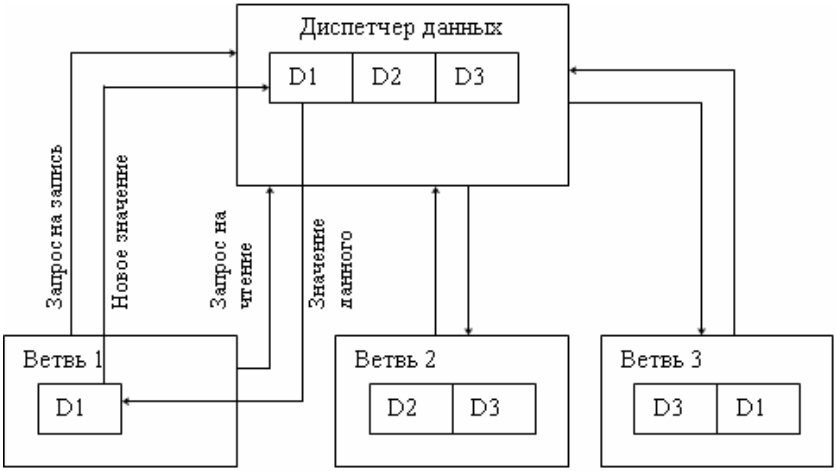


Рис. 4.5. Модель общих данных в технологии ГСП

4.4.2.1 Реализация свойств

Свойство — способ доступа к внутреннему состоянию объекта, имитирующий переменную некоторого типа. Обращение к свойству объекта выглядит так же, как и обращение к полю объекта, но в действительности оно реализовано через вызов функции. При попытке задать значение данного свойства вызывается один метод (setter), а при попытке получить значение данного свойства — другой (getter).

Реализацией свойств в технологии ГСП являются шаблонные классы __property_rw и __indexed_property_rw. В качестве параметра шаблону передается тип переменной, которая будет доступна через данное свойство, и ссылка на класс, в котором это свойство будет использоваться, в нашем случае это TPOData. После инициализации объекта свойства ему необходимо установить методы доступа. В качестве методов доступа используются соответствующие методы из класса TPOData.

Для простых типов и для массивов используются различные классы свойств. Для простых типов данных необходимо указать по

одному методу для получения значения и для его установки. Методы должны иметь следующую сигнатуру:

_T (TPOData:: *)(void); //для getter

_T (TPOData:: *)(_T const &); // для setter ,

где _T – тип переменной.

Для массивов необходимо указать 2 метода получения значения и 2 метода установки значения. Одна пара методов используется для получения всего массива, другая пара используется для получения одного элемента по индексу.

_T* (TPOData:: *)(void); // для получения всего массива

_T* (TPOData:: *)(_T* const &); // для установки всего массива

_T (TPOData:: *)(_I const &); // для получения одного элемента по индексу

_T (TPOData:: *)(_I const &, _T const &); // для установки одного элемента по индексу

Это один из вариантов реализации свойств в языке C++, спроектированный с учетом особенностей технологий ГСП и MPI.

4.4.2.2 Доступ к данным

Класс TPOData и экземпляр класса напрямую недоступен для разработчика граф-программ, и разработчик не обязан знать о его существовании. Когда разработчик обращается к переменной ПОП, на самом деле он обращается к полям-свойствам объекта D. Рассмотрим использование свойств на примере создания актора.

Пусть в ПОП определены следующие типы и переменные (табл. 4.1).

Таблица 4.1

Типы			**Переменные**	
Название	Определение	Описание	Название	Определение
int		Встроенный тип целых чисел	A	int
double		Встроенный тип чисел с двойной точностью	B	double
array	typedef int array[100]	Массив целых чисел длины 100	C	array

Пример доступа к переменным:

a = 100; //устанавливает значение a в 100,

double _b = b; //читает значение b в локальную переменную _b,

b+=1; //инкрементирует значение b,

for (int i = 0; i < c.length; i++) c[i] = i; //инициализирует каждый элемент массива.

В примере показана дополнительная возможность массивов – это получение их длины как значения поля length.

4.4.3 Использование локальных переменных в параллельных процессах

Для повышения производительности вычислений на распределенной системе (например, на кластере), когда скорость доступа к данным в локальной памяти вычислительного узла существенно отличается от скорости доступа к данным другого узла, необходимо увеличивать долю локальных вычислений. Для этого необходим механизм копирования данных из менеджера памяти в локальную память узла, на котором исполняется параллельная ветвь программы. Таким механизмом может служить механизм передачи сообщений между параллельными ветвями, основным назначением которого является синхронизация параллельных ветвей. Вместо передачи единичного флага, разрешающего запуск конкретной вершины, сообщение может содержать данные предметной области. Дуги синхронизации помечаются данными, пересылаемыми в каждом сообщении. При создании сообщения определяется список переменных для отправки и список соответствующих им переменных для приема. Если переменные векторные, то может быть задан диапазон индексов отправляемых и принимаемых элементов вектора. При этом необходимо обеспечить соответствие размера и типов отправляемых и принимаемых данных.

Множество данных предметной области разбивается на два подмножества: общие данные и локальные данные. Общие данные создаются и инициализируются менеджером памяти. Локальные данные создаются и инициализируются в том процессе, в котором они используются. Принадлежность данных конкретному подмножеству определяется установкой флага «Локальное данное» в свойствах ка-

ждого данного. По умолчанию все данные – общие. В передаваемых между вершинами сообщениях в качестве передаваемых и/или принимаемых данных должны участвовать локальные данные.

4.3 Знакомство с технологиями MPI и OpenMP

Несмотря на существование множества различных технологий и средств параллельного программирования, на момент написания настоящего пособия в области параллельных вычислений доминируют две технологии: MPI и OpenMP [3].

MPI используется в основном при разработке параллельных программ для систем с распределенной памятью (массивно-параллельных систем, кластеров и GRID-систем), в то время как OpenMP – для систем с общей памятью (SMP-систем).

Существуют реализации указанных технологий для различных языков программирования и различных аппаратных платформ, что позволяет при написании программы максимально абстрагироваться от особенностей вычислительной системы, на которой она будет исполняться.

4.3.1. Технология OpenMP

В технологии OpenMP вычисления распределяются между несколькими потоками, имеющими равноценный доступ к общей области памяти. Для передачи данных между потоками не требуется специальных конструкций. Достаточно использовать для чтения и записи данных различными потоками область памяти с одним и тем же адресом.

Технология OpenMP формирует так называемый «пульсирующий» параллелизм. Выполнение программы начинается в одном главном потоке, называемом «мастер-поток». В некоторой точке мастер-потока может быть порожден набор других потоков, при этом вычисления, описываемые программой, распределяются между порожденными потоками и мастер-потоком и выполняются параллельно. Затем порожденные потоки завершаются и вычисления продолжаются в мастер-потоке (рис. 4.6).

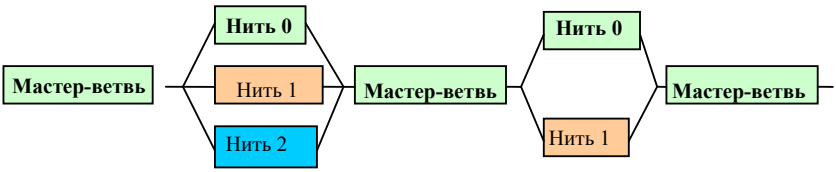


Рис. 4.6. Модель «пульсирующего» параллелизма технологии OpenMP

Каждый поток также может породить набор потоков, после завершения которых вычисления продолжатся в данном потоке.

Для описания параллельных фрагментов программы технология OpenMP использует директивы компилятора вида

```
#pragma omp <имя_директивы> [clause,…]
```

Таким образом, для успешного применения технологии необходимо, чтобы компилятор ее поддерживал. Существует множество компиляторов различных производителей, а также компиляторов с открытым исходным кодом (Open Source), поддерживающих данную технологию. Если технология не поддерживается компилятором, то ее директивы будут игнорироваться. Это позволяет в некоторых случаях создавать универсальный исходный код программы, компилируемый как параллельный или последовательный в зависимости от имеющегося компилятора.

При наличии соответствующего компилятора программа может быть написана на любом языке программирования. На практике для написания программ вычислительного характера используются языки C и Fortran.

Выделяют следующие преимущества технологии OpenMP:

– возможность поэтапного распараллеливания программы путем вставки директив OpenMP в ее различные участки без изменения структуры и основного исходного кода программы;

– возможность создавать программы с универсальным исходным кодом, компилируемые как последовательные или параллельные в зависимости от используемого компилятора;

– возможность использования единого способа распараллеливания вычислений в программах на различных языках программирования, если компилятор соответствующего языка поддерживает технологию OpenMP;

– возможность написания параллельных программ без учета особенностей вычислительной системы – оптимальная реализация директив OpenMP на конкретной платформе выполняется в этом случае разработчиками платформы и обеспечивается предоставляемым ими компилятором.

4.3.2. Технология MPI

Технология MPI (Message Passing Interface) применяется для создания программ, работающих в системах с распределенной памятью. Каждый процесс параллельной MPI-программы имеет собственную область памяти. Обмен данными между процессами выполняется явными операциями пересылки данных в виде сообщений. Технология MPI определяет перечень, семантику и синтаксис функций пересылки сообщений. Спецификация функций не привязана к конкретным языкам программирования, поэтому обеспечивается переносимость программ.

Фактически реализация MPI для конкретной системы – это библиотека функций на некотором языке программирования и набор программ, обеспечивающих среду выполнения MPI-приложений. Наиболее распространены реализации библиотек MPI для языков C и Fortran. Как правило, производители вычислительных систем предоставляют библиотеки MPI, оптимизированные для выполнения на конкретной вычислительной системе. Существуют и свободно распространяемые версии библиотек MPI.

Среда выполнения MPI-приложений обеспечивается программами-«демонами», работающими на всех узлах вычислительной системы и осуществляющими запуск вычислительных процессов на конкретных узлах. Кроме того, в состав среды выполнения часто входит набор утилит для ее конфигурирования и мониторинга.

MPI-приложение состоит из одного исполняемого файла, который копируется и запускается на заданном числе узлов вычислительной системы. Таким образом, на различных узлах запускается одна и та же программа. Среда выполнения присваивает каждому экземпляру программы уникальный номер, называемый рангом процесса. В составе библиотеки MPI имеется функция MPI_Comm_rank для определения ранга процесса, в котором вызвана эта функция. Путем вызова функции MPI_Comm_rank отдельный процесс может опреде-

лить свой ранг и в зависимости от его значения выполнить уникальную последовательность действий. Типичная MPI-программа содержит набор операторов условного перехода, обеспечивающих выполнение различных фрагментов программы (параллельных ветвей) для различных значений ранга текущего процесса.

4.4. Структурное преобразование модели параллельных вычислений для систем с распределенной памятью MPI

Быстрое развитие многопроцессорных вычислительных систем кластерного типа с распределенной памятью привело к необходимости не столько к решению задачи распределения вычислительной нагрузки параллельного алгоритма, но в большей степени, к необходимости разрешения проблемы организации информационного взаимодействия между процессорами при передаче данных. Решение этих проблем обеспечивает интерфейс передачи данных MPI. В рамках MPI проблема эффективной организации параллельных вычислений неизбежно связана с решением проблемы рациональной организации передачи данных между параллельными процессами. Неслучайно, что подавляющее большинство функций стандарта MPI посвящено передаче данных между процессами.

Более естественным для решения задачи построения алгоритмов параллельных вычислений представляется использование многопроцессорных вычислительных систем с общей памятью (симметричных мультипроцессоров SMP), где отсутствует необходимость организации управления данными. Графическая модель алгоритмов параллельных вычислений, созданная на основе модели ГСП-технологии, в первую очередь ориентирована на подобную конфигурацию.

Для систем с общей памятью при разработке модели параллельных вычислений в среде PGRAPH на структуру граф-программы не накладывается сколь-нибудь серьезных ограничений. Однако в стандарте MPI возникает необходимость структурных преобразований граф-программы к виду, приемлемому для использования данного стандарта.

Рассмотрим пример граф-модели параллельного алгоритма, представленного на рис. 4.7.

Как говорилось ранее, для описания параллелизма вводится понятие **параллельной ветви** β – подграфа графа G, начинающегося

параллельной дугой и заканчивающегося терминирующей дугой. $\beta = <A\beta, \Psi\beta>$, где $A\beta$ – множество вершин ветви, $\Psi\beta$ – множество дуг управления ветви.

В соответствии с определением, каждая ветвь имеет ровно один вход и один выход. Число параллельных ветвей в модели фиксируется при ее построении, при этом максимальное количество ветвей не ограничивается. При кодировании алгоритма, описанного с помощью предлагаемой модели, каждая параллельная ветвь порождает отдельный процесс – совокупность операторов, исполняемых последовательно на одном из процессоров параллельной вычислительной системы.

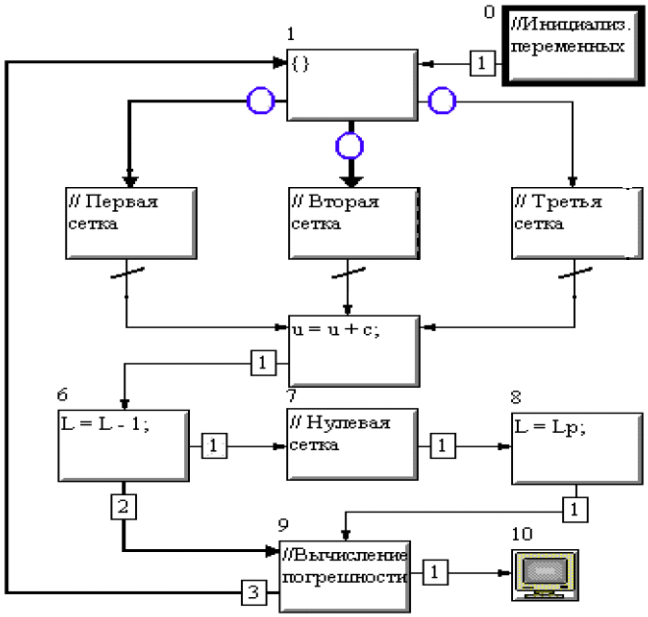


Рис. 4.7. Граф-модель параллельного алгоритма

Вершины, из которых исходят параллельные дуги, будем называть вершинами ветвления и на схемах обозначим их символом «Н». Вершины, в которых «сходятся» терминирующие дуги, назовем тер-

минирующими вершинами и обозначим символом «E». Вершины, принадлежащие параллельным ветвям, обозначим символом «#».

Введем понятие *правильно построенного* агрегата (ППА), схематично представленного на рис. 4.8, где элементами *1* и *2* обозначены последовательные подграфы, имеющие дуги только первого типа. Правильно построенный агрегат имеет только одну вершину ветвления, одну терминирующую вершину и множество параллельных вершин, связанных с вершинами ветвления и терминации.

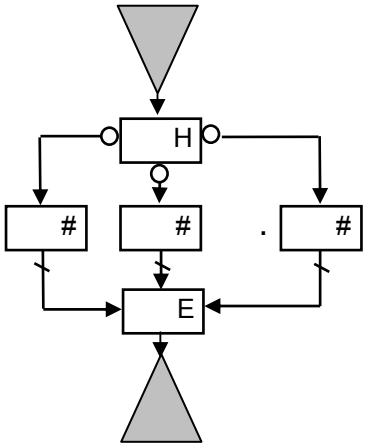


Рис. 4.8. Правильно построенный агрегат

Введение понятия ППА вызвано особенностями организации универсальной граф-машины, построенной с учетом специфики стандарта MPI.

К правильно построенному агрегату отнесем граф, составленный из последовательного соединения ППА, который можно назвать «гирляндой» или многофазным агрегатом.

Очевидно, что при разработке модели параллельных вычислений пользователь не должен руководствоваться введенными ограничениями, связанными с особенностями организации граф-машины в MPI. Поэтому исходная модель организации параллельных вычислений может быть построена произвольным способом в соответствии с правилами формирования агрегатов, принятых в системе PGRAPH, например, так как это показано на рис. 4.9.

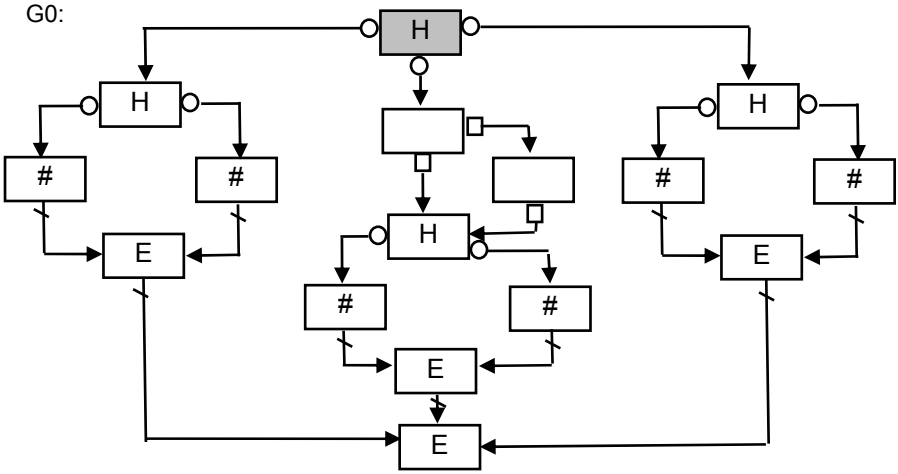


Рис. 4.9. Исходный агрегат

Мастер-ветвь агрегата G0 содержит 3 параллельные ветви, в каждую из которых вложены еще по две параллельные ветви, и в этом смысле G0 не является ППА. Для выполнения параллельных вычислений в системе PGRAPH необходимо произвести разбиение структуры графа G0 на совокупность правильно построенных агрегатов, например, как это показано на рис. 4.10.

Можно сформулировать следующую формальную постановку задачи:

Пусть задан ориентированный граф $G_0 = (A, \Psi), |A| = n$ и пусть

$$\widetilde{A} = \{A_1, ..., A_n, G_1, ..., G_r\}.$$

Требуется найти разрезание $R(\widetilde{A}) = (\widetilde{A}^{(1)}, \widetilde{A}^{(2)}, ..., \widetilde{A}^{(r)})$.

Здесь $\widetilde{A}^{(i)}$ – подмножество множества вершин $\widetilde{A}$ графа G_0 на r *правильно построенных подграфов* G_i, таких что $\widetilde{A} = \bigcup_{k=1}^{r} \widetilde{A}^{(k)}, \widetilde{A}^{(i)} \bigcap \widetilde{A}^{(j)} = \varnothing, i \neq j$.

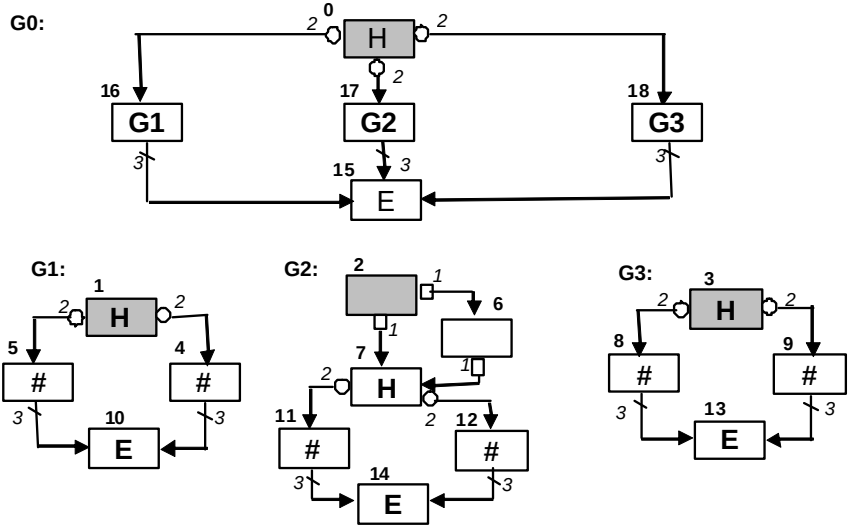


Рис. 4.10. Разбиение агрегата G0 на правильные компоненты

Описание алгоритма

Поставленная задача предполагает выполнение предварительного анализа графа с целью изучения его строения и выявления требуемых подструктур с уточнением инфраструктуры графа относительно выделенных подструктур. Обычно этим занимается теория нумераций графов [21].

Вершины графа будем помечать метками, имеющими следующую семантику:

$$M = < N_{ur} . Type . N_1 . N_2 . \cdots . N_{Nur} > ,$$

где N_{ur} – количество уровней иерархии при вложенности параллельных структур друг в друга; N_i – номер вершины на i-м уровне иерархии (если *i=Nur*) или номер иерархии вложения; *Type* – тип вершины (*Type=H* для вершины ветвления; *Type=E* для терминальных вершин; для остальных *Type=V*).

С помощью алгоритма поиска в глубину [21] на первом этапе мы получаем разметку вершин графа, представленную на рис. 4.11. При этом использовалось описанное выше представление графа с помощью структур смежности графа на смежной памяти (массивы ListTop и ListGraph), которое автоматически получается после предварительной обработки исходного графа G0.

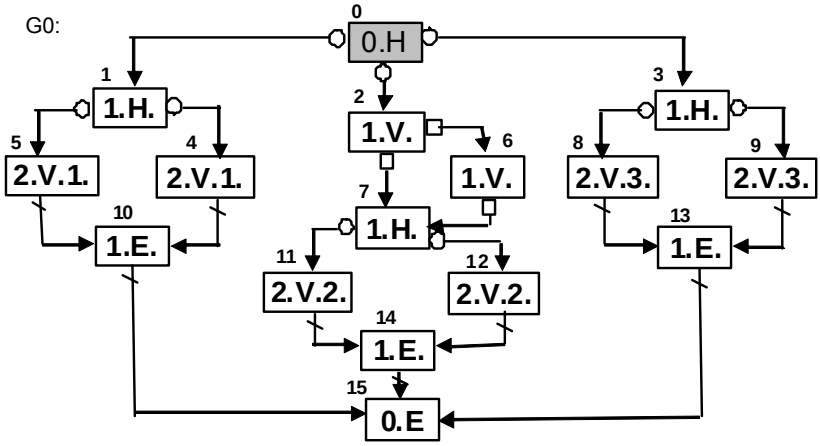


Рис. 4.11. Размеченный граф

Несложный анализ меток вершин разрезаемого графа позволяет определить число правильно построенных и вложенных компонент графа. На самом деле, количество ППА равно числу вершин типа «H» минус 1 (мастер ветвь G0 не в счет). Для нашего примера, где тип «H» имеют вершины <0.H>, <1.H.1>, <1.H.2> и <1.H.3>, необходимо построить как минимум 3 правильные структуры: G1, G2 и G3.

Следует отметить, что если в графе между метками одного уровня, например <1.H.1> и <1.E.1>, на параллельных ветках встречаются последовательно расположенные вершины с одинаковыми метками,

например <2.V.1.1>, то их необходимо агрегировать в одну вершину. Новые вершины дописываются в конец массива ListTop.

На втором этапе работы алгоритма несложными преобразованиями массивов ListTop и ListGraph формируется мастер-ветвь алгоритма G0. Для оформления ППА, вложенных в мастер-ветвь G0, необходимо для каждого агрегата (Gi) найти корневую вершину, которая является смежной вершине <0.H>, и концевую вершину, стоящую в данной параллельной ветке перед вершиной <0.E>. При этом входная вершина помечается как корневая, а для концевой вершины ППА в массиве ListTop записывается соответствующий код (-77). Аналогично обрабатываются параллельные структуры, вложенные в любые параллельные ветки исходного графа.

Предложенный алгоритм структурного преобразования модели параллельного алгоритма, представленного графом управления, позволяет без участия разработчика программы произвести все необходимые структурные преобразования графа управления модели к виду, пригодному для реализации вычислений в среде MPI. Можно показать, что алгоритм имеет линейную сложность $O(n)$, где n – число вершин в графе.

4.5 Использование PGRAPH для разработки алгоритма параллельной глобальной оптимизации

Проблема разработки эффективных методов решения задачи глобальной оптимизации функций многих переменных является чрезвычайно важной для науки и техники. В этой области накоплен значительный опыт и разработано большое количество алгоритмов и методов решения задач многоэкстремальной оптимизации. В то же время преодоление принципиальной трудности задач глобальной оптимизации, связанной с экспоненциальным ростом их сложности в зависимости от размерности оптимизируемой функции, остаётся открытой проблемой, во всяком случае, для точных методов оптимизации.

Современные точные методы глобальной оптимизации совершенствуются главным образом для класса Липшицевых функций в двух направлениях: первое связано с разработкой новых критериев

отсева неперспективных областей оптимизируемой функции, второе – с повышением эффективности методов редукции задачи многомерной оптимизации.

В последнее время решение проблемы высокой вычислительной сложности задач глобальной оптимизации связывают с использованием методов параллельных и распределенных вычислений. Однако использование этой относительно «новой» техники вычислений на практике позволяет получить как существенное ускорение алгоритма глобальной оптимизации, так и замедление его характеристик. Как для любого «нового дела», процесс распараллеливания алгоритмов глобальной оптимизации имеет свои «подводные камни», особенности и специфические трудности. В данном разделе на примере модифицированного метода половинных делений рассматриваются некоторые особенности организации параллельных алгоритмов глобальной оптимизации.

Рассмотрим задачу безусловной глобальной оптимизации непрерывной функции $f : \mathbf{R}^n \to \mathbf{R}$, заданной на допустимом множестве $X \in \mathbf{R}^n$ в следующей постановке:

$$f_* = glob \min_{x \in X} f(x) = f(x_*). \qquad (4.1)$$

Положим, что глобальный минимум x_* принадлежит множеству X_*, причем $X_* \subset X$, а $X = \overset{n}{\underset{i=1}{\otimes}} [0,1]$, является многомерным единичным гиперкубом. Предлагаемый далее алгоритм глобальной оптимизации является модификацией известного метода половинных делений (метода неравномерных покрытий) [21].

Алгоритм неравномерных покрытий гарантирует нахождение ε-оптимального решения, что подтверждается соответствующей теоремой.

Предлагаемая здесь модификация метода половинных делений связана с введением понятия *области притяжения локального минимума* и использованием локальной техники. Пусть $x_i^*, i = 1,..,m$ – локальные минимумы функции $f(x)$. Тогда область притяжения локального минимума x_i^* определим как множество точек начальных

приближений алгоритма локальной оптимизации, из которых алгоритм сходится к x_i^*:

$$S_i = \{x \in X : \arg\min f(x) = x_i^*\}. \quad (4.2)$$

Поскольку построение области притяжений в смысле (4.2) достаточно сложная задача, далее будем использовать следующее упрощенное определение области притяжения локального минимума:

$$S_i = \{x \in X : \| x - x_i^* \| < \rho_i, \quad \rho_i > 0\}, \quad (4.3)$$

где ρ_i – радиус области притяжения локального минимума.

Одной из эффективных стратегий совершенствования алгоритмов глобальной оптимизации (ГО) является использование *локальной техники*, когда стратегия глобального поиска удачно сочетается с методами локальной оптимизации (ЛО).

При ограниченном количестве минимумов функции $f(x)$ области притяжения S_i имеют значительные размеры. Предположим, что относительно оптимизируемой функции известно количество локальных минимумов – r, включая глобальный, и размеры областей притяжения: $\rho_1,...,\rho_r$. Введем понятие «гарантированного радиуса» области притяжения минимумов функции, под которым будем понимать $\rho_m = \min \rho_i, \quad 1 < i \le r$.

В этом случае, как показано в работе [22], среднюю сложность алгоритма глобальной оптимизации можно оценить величиной

$$\widetilde{N}(n) = 2\left(1/4\rho_m\right)^n - 1 + r\log_2(\rho_m/\varepsilon). \quad (4.4)$$

С учетом (4.4), при $\rho_m >> \varepsilon$ этап глобальной оптимизации, связанный с поиском областей притяжения локальных минимумов функции, можно производить достаточно грубо, а следовательно, с меньшими затратами времени.

Дополнительная информация о размерах областей притяжения локальных минимумов функции позволяет построить двухфазный алгоритм метода половинных делений (ДАМПД) глобальной оптимизации, в которой выделим фазы глобальной и локальной оптимизации (рис. 4.12).

В ДАМПД фаза глобальной оптимизации реализуется с помощью модифицированного алгоритма, с той лишь разницей, что двоичное

деление параллелепипедов реализуется до достижения заданных размеров их радиусов R_i. Величина R_i определяется размером «гарантированного радиуса» ρ_m областей притяжения минимумов функции.

На рис. 4.9 пунктирными линиями отмечены гарантированные области притяжения минимумов функции. Заштрихованные прямоугольники являются прямоугольниками, «отбракованными» с использованием константы Липшица. В итоге, исходя из условия $R_i \le \rho_m$, фаза глобальной оптимизации завершилась, породив для приведенного на рис. 4.12 примера 24 прямоугольника заданного радиуса.

В фазе локальной оптимизации из точек, принадлежащих областям притяжения локальных минимумов, организуется поиск минимумов функции с помощью локальных алгоритмов оптимизации. Такая схема организации процедуры поиска глобального минимума функции существенно сокращает трудоемкость фазы глобальной оптимизации.

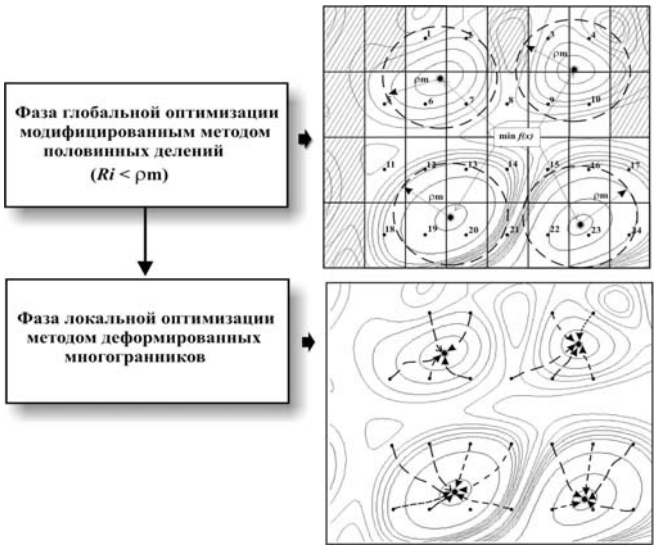


Рис. 4.12. Двухфазный алгоритм метода половинных делений

Для больших размерностей вектора независимых переменных оптимизируемой функции в фазе глобальной оптимизации порождается значительное количество параллелепипедов заданного размера. В этом случае объем вычислений в фазе локальной оптимизации становится чрезмерно большим. Рассмотрим следующий адаптивный алгоритм формирования списка точек начальных приближений областей притяжения локальных минимумов оптимизируемой функции, осуществляющий сжатие количества точек начальных приближений, используемых во второй фазе оптимизации.

Будем считать, что область притяжения S_i определена, если она содержит, по крайней мере, одну точку из множества $C = \{c_1,...,c_k\}$, где c_k – центры параллелепипедов.

Пусть r – количество зон притяжения минимума функции; ρ_m – гарантированный радиус области притяжения минимума функции, обеспечивающий нахождение всех локальных минимумов.

Сформируем список областей притяжения локальных минимумов $V = \{V_1, V_2,..., V_m\}$, элементами которого являются структуры $V_i = (\tilde{c}_i, \tilde{f}_i)$, где $\tilde{c}_i$ – координаты «представителя» i-й области притяжения, имеющего наилучшую, достигнутую для этой области, оценку $\tilde{f}_i$. Вектор $\tilde{c}_i$ условно считается центром i-й области притяжения.

Первоначально список V пуст. По мере вычислений функции он наполняется элементами, но в конце этапа глобального поиска не может содержать больше m элементов (m – заданный размер списка, $m \geq r$). Размер списка m зависит от свойств оптимизируемой функции и выбирается из соображений попадания в него представителя области притяжения глобального минимума функции. Элементы списка V упорядочены таким образом, что $\tilde{f}_1 < \tilde{f}_1 < ... < \tilde{f}_m$. Для построения модели алгоритма ДАМПД достаточно знать, что этот список так или иначе строится и что в первой сотне элементов списка содержится начальное приближение глобального минимума функции.

Реализация алгоритма ДАМПД представлена на рис. 4.13. Для удобства описания модели все вершины пронумерованы.

В вершине *1* устанавливаются значения параметров тестовых функций пакета GKLS и производится его инициализация. Алгоритм

состоит из двух фаз. Первая фаза глобальной оптимизации состоит из вершин 2-5. В вершине *2* формируется начальный список параллелепипедов, которые впоследствии «раздаются» по процессорам кластера (см. вершину *3*).

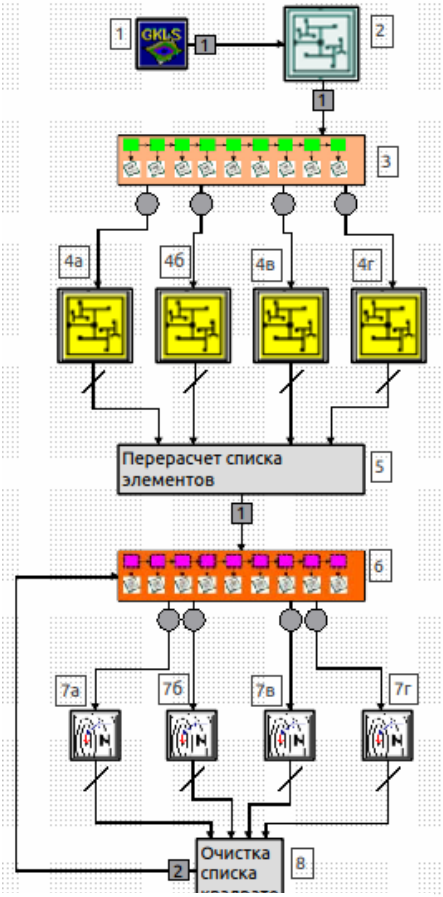


Рис. 4.13. Параллельная версия алгоритма ДАМПД

Глобальная оптимизация методом половинных делений реализуется в вершинах *4а-4г*. На рис. 4.10 для наглядности приведён вариант алгоритма для четырех процессоров, хотя несложно его масштабировать на любое число процессоров. Параллелепипеды делятся до тех пор, пока их радиус не станет меньше ρ_m. При этом на каждом процессоре формируются списки точек, лежащих в областях притяжения локальных минимумов. В вершине *5* сформировавшиеся списки объединяются.

Во второй фазе алгоритма осуществляется поиск локального минимума из точек, вычисленных на первом этапе. Данная фаза состоит из вершин *6-8*. В вершине *6* происходит «раздача» точек начальных приближений процессорам для организации поиска локальных минимумов, который реализуется в вершинах *7а-7г*. Вершина *8* завершает фазу локальной оптимизации. Алгоритм ДАМПД завершает свою работу по исчерпанию списка точек начальных приближений.

Представленный алгоритм отличается простотой, поскольку параллельные ветви программы не обмениваются информацией, и отпадает необходимость реализовывать синхронизацию параллельных вычислений. В то же время разработчик алгоритма освобождается от необходимости использования директив стандарта MPI и может сосредоточиться на самом алгоритме. Визуальная форма алгоритма ДАМПД достаточно наглядна, выразительна и в то же время компилируется системой PGRAPH в исполнимый код.

Тестирование алгоритма ДАМПД проводилось на наиболее сложном для глобальной оптимизации классе недифференцируемых функций. Для всех классов задач: число экстремумов равно десяти; радиус притяжения глобального оптимума – 0,33. Эксперименты проводились на суперкомпьютерном кластере СГАУ «Сергей Королев». Кластер построен на базе линейки оборудования IBM BladeCenter с использованием блейд-серверов HS22 и обеспечивает пиковую производительность более десяти триллионов операций с плавающей точкой в секунду. Общее число процессоров/вычислительных ядер: 272/1184. Глобальный минимум вычислялся с точностью $\varepsilon = 1{,}0 \cdot 10^{-8}$ (по аргументам функции).

Результаты вычислительного эксперимента для алгоритма ДАМПД (при n=8) представлены на рис. 4.14 и 4.15, где показана загрузка процессоров (количество обращений к оптимизируемой функции на каждом из них).

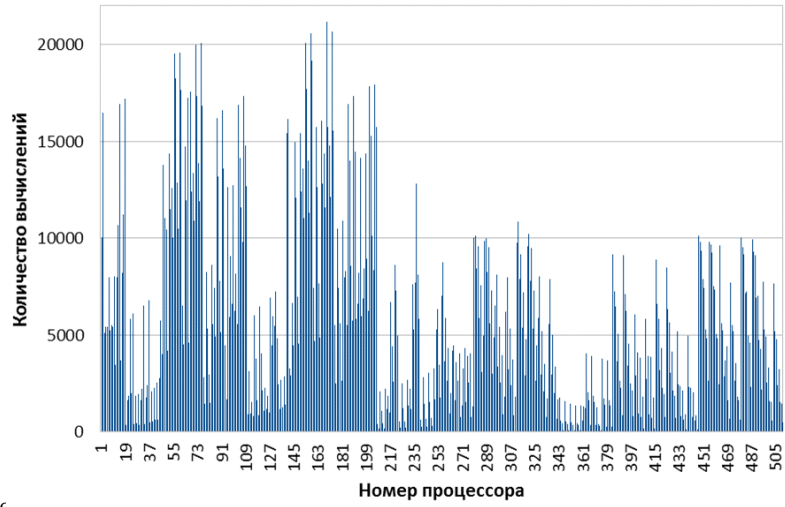


Рис. 4.14. Загрузка процессоров на этапе ГО в базовой версии

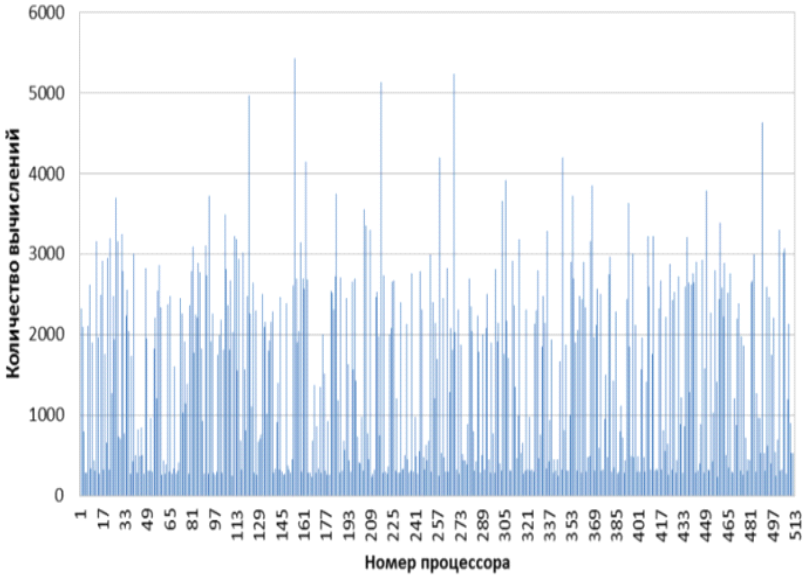


Рис. 4.15. Загрузка процессоров на этапе ЛО в базовой версии

4.6 Краткий обзор

В главе рассматривались концепции визуального, графического способа представления алгоритмов параллельных вычислений, естественным образом полученные за счет расширения концепций описания последовательных алгоритмов, принятых в технологии ГСП. Для параллельного алгоритма его графическая форма записи позволяет в явном виде изобразить организацию параллельных вычислений, что существенно облегчает восприятие алгоритма человеком.

В теоретическом плане для организации параллельных вычислений алгоритмическая модель системы GRAPH расширена

дополнительными компонентами и в первую очередь за счет расширения типов дуг, когда к обычной последовательной дуге были добавлены параллельная, терминирующая и дуга синхронизации.

Существенно изменились и алгоритмы управления вычислительными процессами. Однако, что важно, удалось сохранить принцип централизации всех механизмов управления в рамках универсальной программы, называемой граф-машиной.

Естественно, что так же как для всех систем управления параллельными процессами, в новой системе PGRAPH добавлены средства организации синхронизации параллельных вычислений. В технологии ГСП используется оригинальный комбинированный способ синхронизации, использующий одновременно механизмы передачи сообщений и принцип мониторной синхронизации.

С учетом использования для реализации параллельных вычислений вычислительных систем кластерного типа с распределенной памятью неизбежно приходится использовать протоколы передачи данных между процессорами, например, систему MPI. В системе PGRAPH средства визуального представления алгоритмов предоставляют широкие возможности для наглядного представления моделей параллельных алгоритмов, которые вступают в серьезные противоречия со стандартами системы MPI. В связи с этим предложены методы автоматического преобразования исходной модели параллельных вычислений к стандартному виду, приемлемому для системы MPI. Причем все преобразования производятся скрытно от разработчика.

4.7 Контрольные вопросы

1. Какие основные изменения внесены в алгоритмическую модель технологии последовательных вычислений в системе PGARPH?

2. Опишите назначение параллельной и терминирующей дуг.
3. Что такое параллельная ветвь программы?
4. Чем управляются параллельные и терминирующие дуги параллельного алгоритма?
5. Какая ветвь программы называется мастер-ветвью?
6. Что такое семафорный предикат?
7. Особенности технологий и средств MPI и OpenMP.
8. Какой агрегат называется правильно построенным?
9. В чем суть проблемы приведения?

ГЛАВА 5. МЕТОДЫ КОНТРОЛЯ КОРРЕКТНОСТИ МОДЕЛЕЙ ПАРАЛЛЕЛЬНЫХ ВЫЧИСЛЕНИЙ

В технологии ГСП параллельный вычислительный процесс использует общую память, содержащую переменные со значениями данных предметной области.

Минимальным фрагментом граф-программы, который изменяет значения данных, является ее вершина (точнее – оператор, которым помечена вершина). Так как в параллельной граф-программе несколько вершин могут исполняться одновременно, возможна ситуация, при которой одно и то же данное будет одновременно использоваться несколькими вершинами. Если все вершины используют данное для чтения, то такая ситуация не представляет опасности, значение данного может быть передано всем запросившим его вершинам одновременно.

Если все или часть одновременно исполняющихся вершин в ходе своей работы изменяют значение данного (используют его для записи), то такая ситуация является потенциальным источником ошибок.

Отметим, что предикаты не изменяют значения данных ПрОП, они лишь сравнивают их, т.е. используют для чтения. Поэтому в предикатах подобных ошибок не возникает.

Предположим, что граф-модель параллельной программы содержит фрагмент, в котором несколько вершин исполняются одновременно и записывают результат своей работы в одну и ту же переменную. Совместное использование переменных может быть допущено случайно или введено намеренно. Например, программа оптимизации может искать значение оптимизируемого параметра одновременно несколькими методами. В этом случае каждый из методов представляется отдельной параллельной ветвью. Все ветви используют одну и ту же переменную для записи вычисленного значения параметра. Если для вычислителя не имеет значения, каким из методов получен результат, то ошибки не возникает – модель будет считаться корректной.

Предположим теперь, что в той же граф-модели каждая параллельная ветвь использует некоторую переменную – счетчик (например, счетчик итераций).

Следуя привычному образу мышления, разработчик модели может присвоить одно и то же имя переменной–счетчику в каждой из параллельных ветвей. В этом случае, при их одновременном исполнении, значение счетчика будет в произвольном порядке считываться и изменяться различными ветвями. Очевидно, что целостность переменной–счетчика будет нарушена и результат вычислений будет неверным.

Аналогичная ситуация возникает при коллективной разработке, когда различные параллельные ветви модели создаются различными разработчиками. При объединении их в один граф велика вероятность появления одной и той же переменной в нескольких параллельных ветвях.

На практике модели, предназначенные для решения вычислительных задач, как правило, имеют сложную структуру и оперируют большим количеством переменных. Таким образом, поиск в них совместно используемых данных становится непростой задачей. Наилучшим решением является разработка методов автоматизированного поиска таких данных.

В технологии ГСП для каждого объекта модели поддерживается специальная структура данных, называемая паспортом объекта. Паспорт объекта содержит перечень всех данных предметной области, используемых объектом, а также информацию о способе использования каждого данного: чтение или запись. Паспорта объектов вместе с описанием структуры граф-модели хранятся в информационном фонде системы моделирования, что делает удобным их автоматизированную обработку и анализ.

Рассмотрим различные методы поиска критических данных, основанные на информации из паспортов объектов.

5.1 Простейший метод поиска критических данных в модели параллельного вычислительного процесса

Возможность модификации данного в граф-модели вычислительного процесса определяется специальным признаком – **способом использования** данного.

Способ использования определяется формально как функция $H(\alpha|d)$, использующая два операнда:

α – объект граф-модели, в качестве которого может выступать отдельный актор, подграф агрегата (ветвь) или целый агрегат;

d – данное предметной области.

Функция $H(\alpha|d)$ показывает, каким образом данное d используется объектом α.

Вводится следующая семантика для значений функции H:

$H(\alpha|d) = 0$ – d не используется объектом α,

$H(\alpha|d) = 1$ – d используется для чтения объектом α,

$H(\alpha|d) = 2$ – d используется для записи объектом α,

$H(\alpha|d) = 3$ – d – критическое данное (конфликт совместного использования данного d в объекте α).

ОПРЕДЕЛЕНИЕ. Данное *d* называется ***критическим***, если оно используется несколькими вершинами граф-модели, которые в процессе вычислений могут исполняться одновременно. При этом хотя бы одна из этих вершин помечена объектом, использующим данное *d* для записи.

Под одновременным исполнением мы понимаем пересечение отрезков времени, в течение которых исполняются интересующие нас вершины.

Возникает задача выделения из множества вершин граф-модели подмножества таких вершин, которые могут исполняться одновременно и используют для записи одни и те же данные. При решении этой задачи как минимум необходимо первоначально сформировать списки одновременно исполняющихся вершин. Будем называть такие вершины *параллельными*.

Параллельные вершины однозначно должны принадлежать различным параллельным ветвям, однако данное требование не является достаточным – в программе могут присутствовать последовательные участки, содержащие параллельные ветви. Следовательно, чтобы ответить на вопрос, являются ли две вершины параллельными, необходимо внимательно изучить структуру граф-модели, проверяя принадлежность вершин различным параллельным ветвям и рассматривая взаимное расположение этих ветвей относительно друг друга.

Рассмотрим простейший метод поиска параллельных вершин.

ОПРЕДЕЛЕНИЕ. ***Состоянием модели параллельного вычислительного процесса*** в некоторый момент времени t будем называть множество, состоящее из вершин, в которых происходят вычисления в момент времени t: $S_t = \{A_{i_0},..., A_{i_{k-1}}\}$.

Так как в каждой параллельной ветви одновременно может выполняться не более одной вершины, значение k (число одновременно исполняющихся вершин) не может превышать числа параллельных ветвей в модели (обозначим его через K), за вычетом мастер-ветви: $k \le K$-1.

В зависимости от структуры граф-модели, размерность множества S_t в процессе реализации вычислительного процесса может изменяться, сужаясь до одномерного, если вычислительный процесс идет по одной последовательной ветви, или расширяясь до m, если одновременно выполняются m параллельных ветвей.

Для поиска параллельных вершин необходимо построить множество S *всех возможных состояний* модели вычислительного процесса. Отметим, что множество S не описывает состояние модели в любой момент времени, а содержит все возможные варианты различных состояний модели. Подмножество S^p множества S, элементы которого содержат более одной вершины, определяет все сочетания одновременно исполняющихся вершин, доступных в модели, то есть множество подмножеств параллельных вершин. Элементы подмножества S^p будем называть *параллельными состояниями*.

Максимальное количество элементов подмножества S^p равно

$$[S^p]=2^{(n-2)}-(n-2),$$

где n – количество вершин граф-модели.

Действительно, состояния модели можно закодировать двоичным n-значным числом, в котором наличие "1" в i-м знаке обозначает присутствие в состоянии i-й вершины модели, а наличие "0" в i-м знаке – ее отсутствие. Количество различных кодов в такой системе кодирования равно 2^n. Поскольку в любой граф-модели есть начальная вершина и, по крайней мере, одна конечная вершина, исполняющиеся последовательно, для кодирования параллельных состояний модели достаточно $2^{(n-2)}$ двоичных знаков. Из общего количества возможных сочетаний (n-2) вершин модели следует вычесть

число состояний, при которых в модели исполняется только одна вершина. Тогда максимальное количество параллельных состояний $[S^p]=2^{(n-2)}-(n-2)$.

После построения множества S^p поиск критических данных будет заключаться в последовательном просмотре паспортов объектов, которыми помечены вершины, принадлежащие элементам множества S^p, для всех данных предметной области и всех элементов множества S^p. Если некоторое данное используется одновременно для чтения и записи вершинами, принадлежащими одному и тому же элементу множества S^p, то в указанных вершинах может возникнуть конфликт совместного использования этого данного. Граф-модель алгоритма простейшего метода поиска критических данных приведена на рис. 5.1.

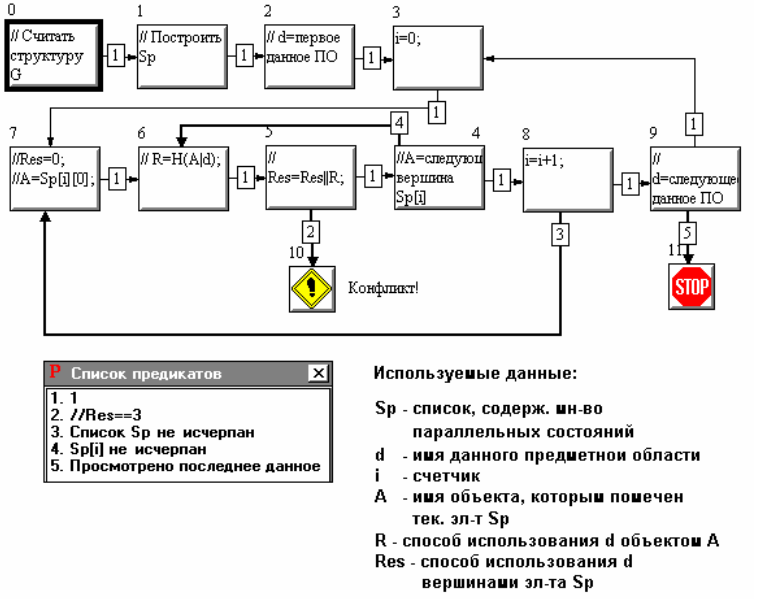


Рис. 5.1. Граф-модель алгоритма простейшего метода поиска критических данных

Оценим временну́ю сложность приведенного алгоритма, под которой будем понимать количество элементарных операций, выполненных алгоритмом.

Для построения множества S^p в оперативной памяти ЭВМ создается копия описания структуры исследуемого агрегата. Поскольку множество S^p строится по этой копии, без обращения к информационному фонду, будем считать, что время, затрачиваемое на построение S^p, мало по сравнению с длительностью исполнения остальной части алгоритма.

В граф-модели на рис. 5.1 максимальную длительность исполнения имеют объекты в вершинах *6* и *9*, так как они содержат обращения к информационному фонду и используют операции с внешними запоминающими устройствами.

Остальные вершины содержат простейшие операции с переменными в оперативной памяти, поэтому будем считать, что длительность исполнения алгоритма определяется в основном вершинами *6* и *9*, а временем работы остальных вершин можно пренебречь. В качестве элементарной операции будем рассматривать операцию обращения к внешнему запоминающему устройству, которая вызывается в вершинах *6* и *9*. Длительности исполнения вершин *6* и *9* примем равными и обозначим их τ.

Определим максимально возможное количество запусков вершин *6* и *9* на исполнение. Алгоритм на рис. 5.1 включает 3 цикла. Обозначим их L1, L2 и L3:

L1 – вершины 6, 5, 4;

L2 – вершины 7, 6, 5, 4, 8;

L3 – вершины 3, 7, 6, 5, 4, 8, 9.

Количество итераций цикла L1 определяется числом вершин в элементах списка Sp. Максимальное число таких вершин равно (n-2), где n – количество вершин исследуемого агрегата G.

Число итераций цикла L2 зависит от длины списка Sp. Выше показано, что максимальная длина списка Sp равна $2^{(n-2)}$-(n-2).

Цикл L3 организован по всем данным предметной области, поэтому число его итераций равно количеству данных. Обозначим его *l*.

Тогда временная сложность алгоритма простейшего метода поиска критических данных (обозначим ее M_1) оценивается выражением (5.1):

$$M_1 \le ((n-2)*(2^{(n-2)}-(n-2))+1)*l. \qquad (5.1)$$

Длительность работы алгоритма (обозначим ее T_1) будет оцениваться следующим соотношением:

$$T_1 \le ((n-2)*\tau*(2^{(n-2)}-(n-2))+\tau)*l.$$

Полученные соотношения будут использоваться ниже по тексту для сравнения с характеристиками алгоритма, реализующего метод поиска критических данных, предложенный автором.

5.2 Метод поиска критических данных на основе алгебры над способами использования данных

Оперируя понятием способа использования, задачу поиска критических данных можно сформулировать следующим образом: зная способ использования данных в каждой вершине, найти такие данные, для которых способ использования граф-модели в целом равен 3.

Рассматриваемый ниже алгоритм поиска критических данных решает задачу именно в такой формулировке. Он основан на построении формального описания граф-модели, которое позволяет обнаруживать в ней критические данные на основе информации о способе использования данных каждой вершиной и взаимном расположении этих вершин.

Перед тем как перейти к описанию алгоритма, сделаем несколько замечаний.

Первое из них касается случаев, когда из одной вершины в другую следует несколько дуг управления. Предположим, что из вершины A_1 в вершину A_2 направлено n дуг, помеченных предикатами p_1, p_2, …, p_n (рис. 5.2).

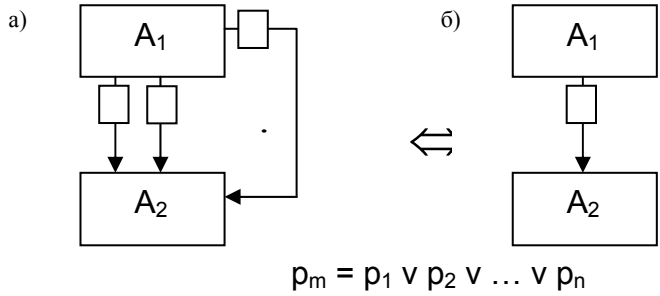


Рис. 5.2. Замена нескольких дуг управления одной дугой

Переход из вершины A_1 в вершину A_2 возможен только при истинности одного или сразу нескольких предикатов $p_1, p_2, \ldots, p_n$. Приоритеты дуг значения не имеют. Очевидно, что n дуг в этом случае могут быть заменены одной дугой, предикат которой будет равен дизъюнкции предикатов $p_1, p_2, \ldots, p_n$. С учетом приведенных рассуждений далее предполагается, что из одной вершины в другую может следовать не более одной дуги.

Второе замечание связано с исследованием иерархических моделей, агрегат которых в качестве вершин содержит другие агрегаты. Для поиска критических данных такие модели могут быть приведены к форме без иерархии. При этом вершины, содержащие агрегаты, заменяются подграфами, соответствующими этим агрегатам.

Замена производится по следующему алгоритму:

1) входящие в заменяемую вершину A_i дуги направляются в корневую вершину агрегата G_k^i, содержащегося в вершине A_i;

2) если в агрегате G_k^i имеется более одной конечной вершины, то в него добавляется фиктивная конечная вершина, не выполняющая никаких действий (с нулевым временем выполнения), в которую направляются дуги из других конечных вершин;

3) дуга, исходящая из вершины A_i, заменяется на дугу из конечной вершины агрегата G_k^i.

Предложенный способ исключения иерархии соответствует принципам управления вычислительным процессом в модели, по-

этому не приводит к потере или появлению новых критических данных. Поскольку фиктивные конечные вершины не обращаются к данным, их введение также не влияет на способ использования данных в модели.

Пример исключения иерархии из модели вычислительного процесса приведен на рис. 5.3.

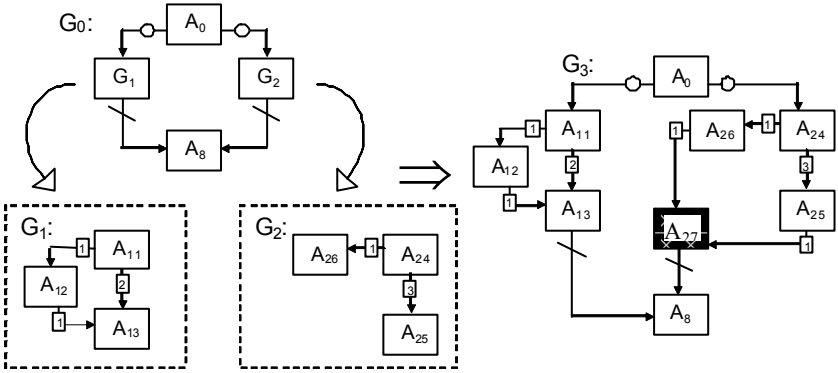


Рис. 5.3. Исключение иерархии из модели вычислительного процесса

Рассмотрим возможные варианты взаимного расположения вершин в граф-модели.

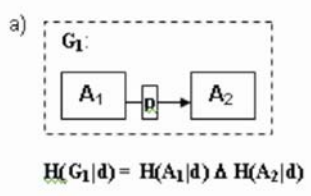
Простейший случай – передача управления из одной вершины модели в другую. Введем двуместную **операцию следования** над способами использования данных в вершинах, соединенных дугой управления. Обозначим эту операцию символом "Δ". Результатом операции будет способ использования данных для подграфа, состоящего их двух вершин и соединяющей их дуги. Пример подграфа приведен на рис. 5.4, а.

Очевидно, что при отсутствии критических данных в вершинах такой подграф не приведет к их возникновению. Вершины подграфа выполняются строго последовательно, поэтому конфликтов при доступе к данным нет.

Критические данные в таком подграфе могут быть только в том случае, когда они уже присутствуют в одной из его вершин, то есть способ использования некоторого данного в одной из вершин подграфа равен 3. Для определения результата операции следования

при других значениях способа использования данных учтем то обстоятельство, что критические данные возникают только при использовании данных для записи. Если рассматриваемый нами подграф будет исполняться одновременно с другой параллельной ветвью, имеющей с ним общие данные, то при использовании этих данных для записи в любой из вершин A_1 или A_2 может возникнуть конфликтная ситуация. Следовательно, результат операции следования должен быть равен 2, если хотя бы одна из вершин подграфа использует некоторое данное для записи. Результаты операции следования для любых значений способов использования данных приведены в таблице истинности на рис. 5.4, б.

Рассмотрим подграф G_2, изображенный на рис. 5.4, в. Этот подграф обозначает ветвление, то есть развитие вычислительного процесса по одному из нескольких путей. Он состоит из вершин, в которые входят последовательные дуги, исходящие из одной и той же вершины (на рис. 5.4, в) эта вершина изображена пунктиром). Введем для таких подграфов **операцию ветвления** над способами использования данных в вершинах, принадлежащих различным направлениям развития вычислений. Обозначим ее символом "∇". Формула вычисления способа использования данных в подграфе G_2 приведена на рис. 5.4, в. Таблица истинности для операции ветвления приведена на рис. 5.4, б.



б)

H(A1|d)	H(A2|d)	Δ	∇
0	0	0	0
0	1	1	1
0	2	2	2
0	3	3	3
1	0	1	1
1	1	1	1
1	2	2	2
1	3	3	3
2	0	2	2
2	1	2	2
2	2	2	2
2	3	3	3
3	0	3	3
3	1	3	3
3	2	3	3
3	3	3	3

Рис. 5.4. Операции следования и ветвления

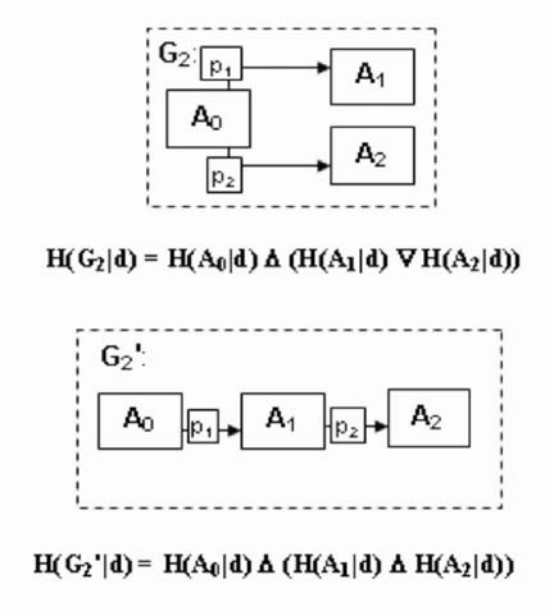


Рис. 5.5. Замена операции ветвления на операцию следования

Как видно из таблицы, результат операций следования и ветвления совпадает. Операция ветвления определена на основе рассуждений, приведенных выше для операции следования. Тем не менее введение операции ветвления удобно при построении формул для вычисления способа использования данных в агрегатах со сложной структурой.

Равенство результатов операций следования и ветвления позволяет заменить в формуле на рис. 5.4, в операцию ветвления на операцию следования:

$H(G_2|d) = H(A_0|d) \Delta (H(A_1|d) \nabla H(A_2|d)) = H(A_0|d) \Delta (H(A_1|d) \Delta H(A_2|d))$.

Тогда подграф G_2 можно заменить подграфом G_2', состоящим из цепочки вершин A_0, A_1, A_2 (рис. 5.5).

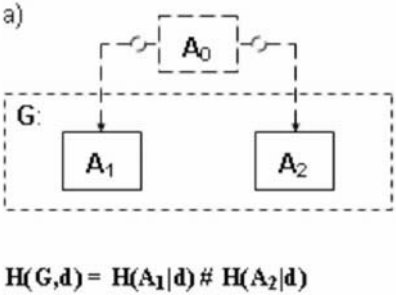
Отметим, что такая замена возможна лишь с точки зрения поиска ошибок совместного использования данных. Подграфы G_2 и G_2' эквивалентны в смысле наличия критических данных, но результаты производимых в них вычислений будут различными.

В дальнейшем при употреблении операций над способами использования данных мы не будем применять рассмотренное свойство, а для упрощения алгоритмов и большей наглядности будем использовать обе операции следования и ветвления.

Подграф, содержащий параллельные вершины, изображен на рис. 5.6, а. Он состоит из вершин, в которые ведут параллельные дуги, исходящие из одной вершины. Для таких подграфов введем **операцию параллельного исполнения** над способами использования данных и обозначим ее символом "#".

Таблица истинности операции параллельного исполнения приведена на рис. 5.6, б. Операция порождает критические данные, если один из ее операндов равен 3, а также в случае, когда оба операнда больше нуля и хотя бы один из операндов равен 2. Другими словами, критическое данное возникает при совместном использовании некоторого данного параллельно исполняющимися вершинами, когда хотя бы одна из этих вершин использует данное для записи.

Используя введенные операции, определим **алгебру над способами использования данных** в граф-модели вычислительного процесса.



б)

$H(A_1\|d)$	$H(A_2\|d)$	#
0	0	0
0	1	1
0	2	2
0	3	3
1	0	1
1	1	1
1	2	3
1	3	3
2	0	2
2	1	3
2	2	3
2	3	3
3	0	3
3	1	3
3	2	3
3	3	3

Рис. 5.6. Операция параллельного исполнения

ОПРЕДЕЛЕНИЕ. ***Алгеброй над способами использования данных*** назовем систему $\Lambda = \langle H, \Omega \rangle$, где $H=\{H(A_1|d_1),\ldots, H(A_1|d_l),\ldots, H(A_n|d_1),\ldots, H(A_n|d_l)\}$ – множество, состоящее из способов использования данных предметной области в вершинах граф-модели, $\Omega = \{\Delta, \nabla, \#\}$ – совокупность операций над способами использования данных.

Правильно построенные формулы во введенной алгебре определим следующим образом.

Элементарной правильно построенной формулой является способ использования $H(A_i|d)$ данного d в некоторой вершине A_i граф-модели, помеченной оператором f_i, а не агрегатом. Кроме того, если α, β – правильно построенные формулы, то следующие формулы будут также правильно построенными:

1) (α),
2) $\alpha \Delta \beta$,
3) $\alpha \nabla \beta$,
4) $\alpha \# \beta$,
5) других правильно построенных формул нет.

Отметим, что предложенная алгебра над способами использования данных содержит формулы, которым нельзя поставить в соответствие корректный в предлагаемой модели агрегат. Так, формуле 3) соответствует агрегат, состоящий из двух отдельных вершин, одна из которых может получить управление. Условие, при котором это происходит, не определено. Поскольку в предлагаемой модели вычислительный процесс начинается с единственного оператора, лежащего в корневой вершине агрегата, рассмотренный случай является некорректным.

Аналогично формула 4) описывает две отдельные вершины, исполняющиеся параллельно и не связанные никакими дугами. Так как параллельные ветви в граф-модели должны начинаться и заканчиваться дугами определенного типа, такой агрегат также будет некорректным. При описании реальных агрегатов формулы 3) и 4) всегда используются совместно с операцией следования. Тем не менее, введение этих формул необходимо для описания случаев, когда из одной вершины исходит более двух дуг, как показано на рис. 5.7.

Для граф-моделей с произвольной структурой, не содержащих иерархии и кратных дуг, алгебра над способами использования данных позволяет строить формулу для вычисления способа использования каждого данного произвольным объектом модели (актором, подграфом граф-модели или целым агрегатом). Если для какого-то данного результат вычисления его способа использования агрегатом равен 3, то в агрегате возможен конфликт совместного использования этого данного.

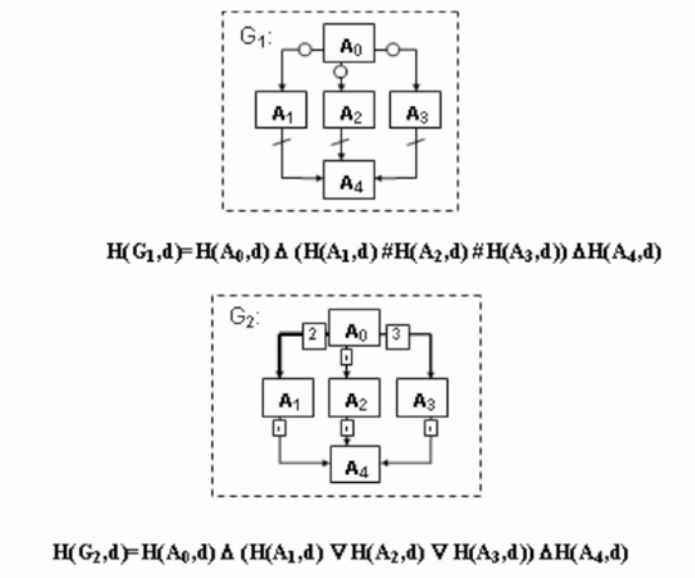


Рис. 5.7. Построение формулы при наличии более двух исходящих из вершины дуг

Рассмотрим свойства операций следования, ветвления и параллельного исполнения (для сокращения записи способ использования данного d в вершине A – H(A|d) – заменим обозначением самой вершины A и опустим «пустые» вершины):

1) AΔB = BΔA (коммутативность операции следования),

2) AΔBΔA = AΔB (поглощение операции следования),

3) $A\Delta(B\Delta C) = (A\Delta B)\Delta C$ (ассоциативность операции следования),

4) $A\Delta B = A\nabla B$ (эквивалентность операций следования и ветвления),

5) $A\#B = B\#A$ (коммутативность операции параллельного исполнения),

6) $A\#(B\#C) = (A\#B)\#C$ (ассоциативность операции параллельного исполнения),

7) $A\#A\#A = A\#A$ (поглощение операции параллельного исполнения),

8) $A\#A \neq A$,

9) $A\#B\#A \neq A\#B$,

10) $A\#(B\Delta C) = (A\#B)\ \Delta\ (A\#C)$, $A\#(B\nabla C) = (A\#B)\ \nabla\ (A\#C)$ (дистрибутивность операции # относительно Δ и ∇),

11) $A\Delta(B\#C) \neq (A\Delta B)\ \#\ (A\Delta C)$,

12) $(A\#B)\ \Delta\ (A\#C)\ \Delta\ (B\#C) = A\#B\#C$.

Приведенные свойства вводятся с учетом семантики операций следования, ветвления и параллельного исполнения и легко проверяются с помощью таблиц истинности этих операций.

Следует обратить внимание на свойство 2 (поглощение операции следования). Это свойство упрощает построение формул для граф-моделей, содержащих циклы. В таких моделях дуги, осуществляющие возврат управления в цикле, могут не учитываться (рис. 5.8).

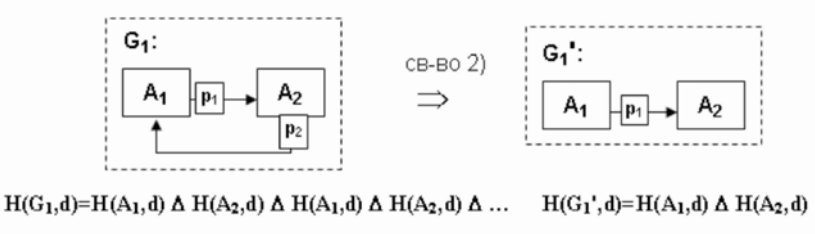


Рис. 5.8. Исключение циклических участков модели при построении формул над способами использования данных

Покажем, что для любой граф-модели можно построить формулу над способами использования данных. Для этого рассмотрим сначала модель, не содержащую параллельных ветвей.

Утверждение 2. *В модели, не содержащей параллельных ветвей, не могут возникнуть критические данные.*

Доказательство

Для критических данных значение способа использования равно 3. Способ использования любого данного может получить это значение только в результате вычисления операции параллельного исполнения. Эта операция определена только для граф-моделей, содержащих параллельные вершины, и принимает значение, равное 3, лишь при использовании данного вершинами различных параллельных ветвей. Следовательно, при вычислении способа использования данного в модели, не содержащей параллельных ветвей, операция параллельного исполнения не встречается, а значит, способ использования данного не может принять значение, равное 3 (критическое данное). Утверждение доказано.

Утверждение 3. *Если заменить способ использования данного $H(A_i,d)$ обозначением вершины A_i, то для любой граф-модели, не содержащей параллельных ветвей, формулу над способами использования данных можно представить в виде перечня вершин граф-модели без повторения, разделенных операцией следования.*

Доказательство

Так как граф-модель не содержит параллельных ветвей, в ее формуле не может присутствовать операция параллельного исполнения. При построении формулы такой граф-модели, на основании свойства эквивалентности операций следования и ветвления, все операции ветвления можно заменить на операции следования. Таким образом, какой бы сложной ни была структура граф-модели, ее формула будет содержать полный перечень вершин модели, символы операции следования, и возможно, скобки. Используя свойство ассоциативности операции следования, скобки из такой формулы можно исключить. Свойство поглощения операции следования позволяет исключить из формулы также все повторные вхождения вершин. После указанных преобразований формула над способами использования данных бу-

дет состоять из перечня вершин граф-модели без повторений, разделенных операцией следования. Утверждение доказано.

Рассмотрим теперь элементарную параллельную граф-модель G_1, содержащую параллельные ветви, но не содержащую последовательных дуг (рис. 5.9).

Граф-модель G_1 можно описать с помощью формулы:

$$H(G_1,d) = H(A_0,d) \Delta (H(A_1,d)\#H(A_2,d)\#\ldots\#H(A_{n-1},d)) \Delta H(A_n,d). \quad (5.2)$$

Предположим, что одна или несколько вершин A_i, i = 1 … n-1, представляет собой агрегат, не содержащий параллельных ветвей. Тогда, согласно Утверждению 2, способ использования данного d вершиной A_i будет определяться формулой

$$H(A_i,d) = H(A^1_i,d) \Delta H(A^2_i,d) \Delta \ldots \Delta H(A^m_i,d), \quad (5.3)$$

где A^r_i, r = 1 … *m* – вершины агрегата A_i.

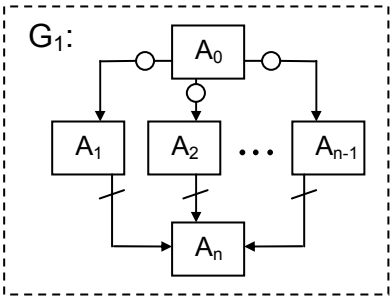


Рис. 5.9. Элементарная параллельная граф-модель

Подставив выражение (5.3) в выражение (5.2), получим способ использования данного d граф-моделью G_1.

Распространим по индукции аналогичные рассуждения на случай граф-модели произвольной сложности, содержащей и последовательные, и параллельные дуги. Такая граф-модель может быть представлена в виде иерархии элементарных граф-моделей, аналогичных приведенной на рис. 5.9, у которых в вершинах A_1, A_2, …, A_{n-1} лежат агрегаты без параллельных дуг. Формула над способами использования данных для такой модели будет комбинацией выражений (5.2) и (5.3).

Таким образом, мы показали, что для любой граф-модели можно построить формулу над способами использования данных. Для этого ее нужно представить в виде иерархии элементарных параллельных граф-моделей, аналогичных приведенной на рис. 5.9, и граф-моделей, не содержащих параллельных дуг.

5.3 Пример применения формулы над способами использования данных для поиска критических данных

Рассмотрим пример, иллюстрирующий применение формул над способами использования данных. На рис. 5.10 приведена граф-модель и соответствующая ей формула.

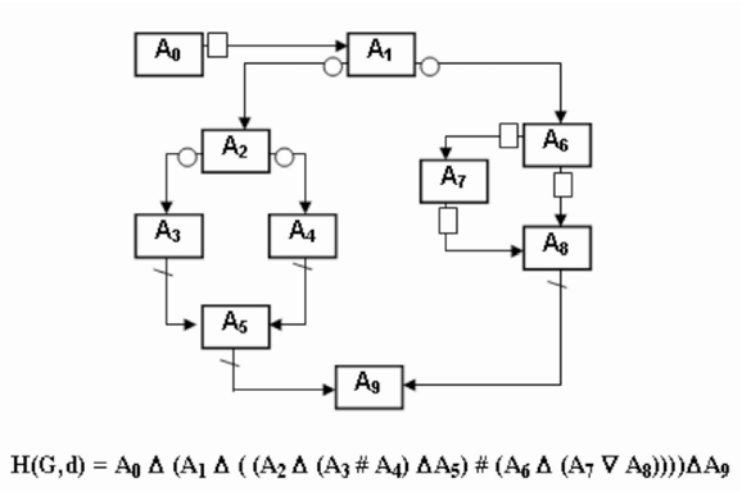


Рис. 5.10. Пример граф-модели и соответствующей ей формулы над способами использования данных

Модель содержит вложенные параллельные ветви, при этом одна из ветвей включает ветвление. Таким образом, приведенная граф-модель демонстрирует применение всех введенных выше операций над способами использования данных.

Пусть вершины A_3 и A_7 производят запись одного и того же данного d. Другие вершины не используют это данное:

$$H(A_3, d)=2,\ H(A_7, d)=2,\ H(A_i, d)=0,\ \forall i,\ i \neq 3,\ i \neq 7. \qquad (5.4)$$

Подставим в формулу рис. 5.10 значения способов использования данного d в вершинах граф-модели:

$$H(G,d) = 0\ \Delta\ (0\ \Delta\ ((0\ \Delta\ (2\ \#\ 0)\ \Delta\ 0)\#\ (0\ \Delta\ (2\ \nabla\ 0)))\ \Delta\ 0. \qquad (5.5)$$

Вычисление формулы (5.5) показывает, что способ использования данного d для граф-модели в целом равен 3 (критическое данное) (см. рис. 5.11).

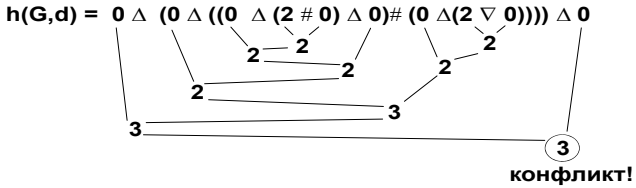


Рис. 5.11. Вычисление способа использования данного d для граф-модели в целом

5.4 Проверка корректности синхронизации граф-модели

Если в граф-модели присутствуют критические данные, разработчику необходимо позаботиться об устранении конфликта путем переименования данных или введения в модель дуг синхронизации. В первом случае каждая параллельная ветвь будет использовать собственное данное, не конфликтуя с остальными участками граф-модели. Во втором случае параллельные ветви будут исполняться согласованно, т.е. не допускать одновременного исполнения участков граф-модели, использующих критическое данное.

Недопустимость одновременного исполнения означает **последовательную работу** модулей с критическими данными. Увеличение доли последовательных вычислений в параллельном алгоритме приводит к снижению его ускорения. Последнее утверждение иллюстрируется следующим соотношением, известным как закон Амдала [3]:

$$S = \frac{p}{\beta \cdot p + (1-\beta)},$$

где S – ускорение параллельного алгоритма; p – количество процессоров в вычислительной системе; β – доля последовательных вычислений в алгоритме.

$\beta = n/N$, где n – число операций алгоритма, выполняемых последовательно, N – общее количество операций.

Принимая во внимание закон Амдала, можно сделать вывод о том, что для повышения производительности вариант переименования данных является более предпочтительным.

На практике не всегда удается распределить данные таким образом, чтобы каждое из них использовалось единственной параллельной ветвью. При этом неизбежно возрастает количество данных, а следовательно, и число работающих с ними модулей, что усложняет работу с моделью. В таком случае необходимо ввести дуги синхронизации, чтобы определить порядок исполнения участков с критическими данными.

Однако введение дуг синхронизации ставит перед разработчиком новые вопросы:

1) Исключают ли введенные дуги синхронизации совместное использование данных?

2) Не приводят ли дуги синхронизации к появлению в программе тупиков?

Описанный выше метод поиска критических данных не дает ответа на указанные вопросы, так как не учитывает дуг синхронизации. Для обнаружения подобных ситуаций разработаны методы, использующие как дуги управления, так и дуги синхронизации.

5.4.1 Метод проверки корректности синхронизации граф-модели

Описываемый ниже метод проверяет, исключают ли введенные в модель дуги синхронизации конфликт совместного использования данных. Поскольку синхронизация должна быть корректной на любом наборе входных данных, метод не учитывает содержимого предикатов, которыми помечены дуги управления в граф-модели. Переход по любой дуге управления считается возможным.

Несколько изменим семантику граф-модели, добавив к каждой дуге управления параллельную ей дугу синхронизации, соединяющую те же вершины, условно положив, что при передаче управления

от одной вершины к другой передается сообщение о разрешении запуска очередного оператора.

Если в граф-модели нет других дуг синхронизации, кроме искусственно добавленных указанным выше образом, то для каждой вершины A_i такой граф-модели можно записать семафорный предикат как дизъюнкцию логических переменных $b_{j,i}$:

$$R(A_i) = (\bigvee_j b_{j,i}),$$

где $j = j_1, \ldots, j_n$ – номера вершин, из которых исходят дуги управления, входящие в A_i. Действительно, вершина A_i может получить управление по любой входящей в нее дуге управления. В этом случае будет сформировано сообщение $\mu_{j,i}$, соответствующее дуге синхронизации, параллельной этой дуге управления, а значит, переменная $b_{j,i}$ будет равна 1.

Если в граф-модели присутствуют другие дуги синхронизации, то семафорный предикат каждой вершины граф-модели можно привести к универсальному виду. Для произвольной вершины A_i семафорный предикат будет иметь следующий вид:

$$R(A_i) = (\bigvee_j b_{j,i}) \wedge (r(b_{k_1,i},...,b_{k_M,i})), \quad j = j_1,...,j_n,$$

где $j_1, \ldots, j_n$ – номера вершин, из которых исходят дуги управления, входящие в A_i; $k_1,\ldots,k_M$ – номера вершин, из которых исходят дуги синхронизации, входящие в A_i; $r(b_{k_1,i},...,b_{k_M,i})$ – логическая функция.

С учетом замечания 4.1, приведенного в разделе 4, согласно которому в семафорном предикате операция "∇" используется только над сообщениями от взаимоисключающих вершин, можно утверждать, что после вычисления семафорного предиката, в момент запуска оператора вершины на исполнение, все вершины, сообщения от которых учитываются при вычислении семафорного предиката, завершили исполнение.

Принятие семафорным предикатом значения истинности в результате его вычисления можно считать условием запуска оператора

вершины на исполнение. Действительно, для того чтобы началось вычисление семафорного предиката некоторой вершины, необходимо, чтобы эта вершина получила управление по дуге управления. После этого единственным условием запуска оператора вершины на исполнение является истинность семафорного предиката данной вершины.

В предлагаемой модели запрещена передача управления между вершинами из различных параллельных ветвей, за исключением тех случаев, когда при такой передаче порождается новая параллельная ветвь или терминируется текущая. Иными словами, две различные параллельные ветви могут быть соединены только параллельными или терминирующими дугами. Отсюда можно сделать вывод, что циклы возможны только в пределах одной параллельной ветви.

Рассматриваемый метод используется для проверки корректного использования критических данных в модели, в которой дуги синхронизации соединяют вершины, не входящие в отличные друг от друга циклы. Это означает, что во время исполнения оператора одной из вершин, соединенных дугой синхронизации, другая вершина не может вновь получить управление. Данное ограничение выполнимо, так как дугу синхронизации всегда можно вынести за пределы цикла, например, проведя ее от вершины, следующей за циклом или предшествующей ему. В таком случае может лишь снизиться быстродействие вычислительного процесса, так как синхронизироваться (ожидать друг друга) будут более крупные его блоки.

Для пояснения сути метода рассмотрим пример (рис. 5.12, а).

Дополнив дуги управления дугами синхронизации, получим граф-модель, изображенную на рис. 5.12, б. Сверху над каждой вершиной написан ее семафорный предикат.

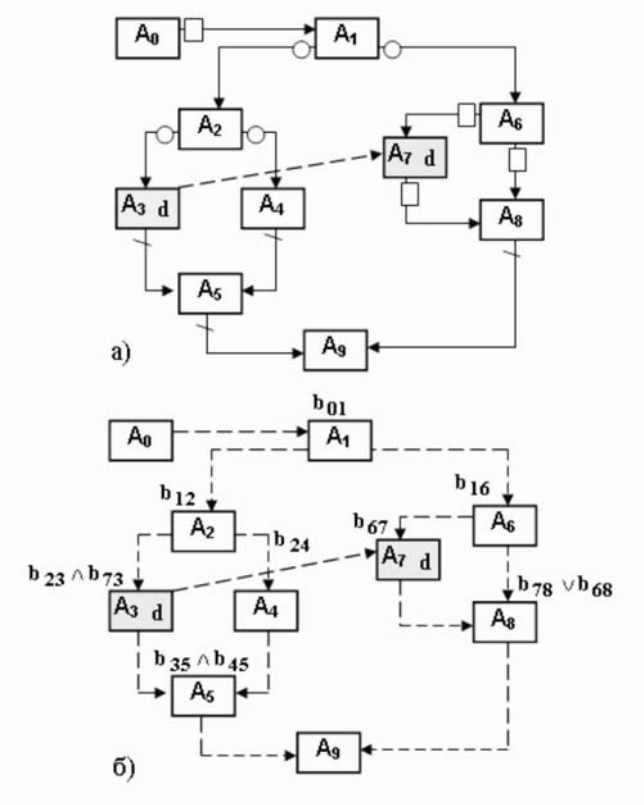


Рис. 5.12. Пример построения обобщенного семафорного предиката

Оператор корневой вершины исполняется безусловно: эта вершина первой получает управление, и ее семафорный предикат R_{A0} тождественно равен 1. Условием запуска оператора в вершине A_1 является истинность семафорного предиката $R_{A1} = b_{01}$. Для того чтобы булева переменная b_{01} приняла значение истинности, необходимо, чтобы в почтовый ящик поступило сообщение μ_{01}, посылаемое от вершины A_0 вершине A_1. Сообщение формируется после завершения исполнения оператора в вершине A_0, а сам оператор, как было указа-

но выше, исполняется после вычисления и принятия значения истинности семафорным предикатом R_{A0}. Тогда необходимым условием истинности R_{A1} является истинность R_{A0}. Таким образом, *в более общем виде* условие запуска на исполнение оператора в вершине A_1 можно представить в виде

$$\widetilde{R}_{A_1} = R_{A_0} \wedge R_{A_1} . \qquad (5.6)$$

Применяя аналогичные рассуждения к вершине A_2, семафорный предикат которой равен $R_{A2} = b_{12}$, получим, что для запуска ее оператора на исполнение необходимо, чтобы оператор в вершине A_1 начал и завершил свое исполнение, а в почтовый ящик поступило сообщение μ_{12}. Условие запуска оператора в вершине A_2 можно записать в следующем виде:

$$\widetilde{R}_{A_2} = \widetilde{R}_{A_1} \wedge R_{A_2} .$$

С учетом выражения (5.6)

$$\widetilde{R}_{A_2} = R_{A_0} \wedge R_{A_1} \wedge R_{A_2} .$$

Последовательно применяя приведенные выше рассуждения ко всем вершинам граф-модели, можно получить условие запуска произвольной вершины A_i.

В рекурсивном виде это условие можно записать так:

$$\begin{cases} \widetilde{R}(A_i) = (\bigvee\limits_{j=j_1}^{j_n} (b_{j,i} \wedge \widetilde{R}_{A_j})) \wedge (r((\widetilde{R}_{A_{k_1}} \wedge b_{k_1,i}),...,(\widetilde{R}_{A_{k_M}} \wedge b_{k_M,i}))), \\ \widetilde{R}(A_0) = 1, \end{cases} \qquad (5.7)$$

где $j_1, \ldots, j_n$ – номера вершин, из которых исходят дуги управления, входящие в A_i ; $k_1,\ldots,k_M$ – номера вершин, из которых исходят дуги синхронизации, входящие в A_i,; $r(b_{k1i}, \ldots, b_{kMi})$ – логическая функция.

ОПРЕДЕЛЕНИЕ. Назовем условие запуска оператора вершины на исполнение, определяемое выражением (5.7), ее **обобщенным семафорным предикатом.**

ОПРЕДЕЛЕНИЕ. ***Дизъюнктивная форма обобщенного семафорного предиката*** представляет собой дизъюнкцию конъюнкций булевых переменных $b_{j,i}$.

Утверждение 4: *Критическое данное d, используемое вершинами A_1, …, A_n, не приводит к конфликту тогда и только тогда, когда для любых двух вершин A_i, $A_j \in \{ A_1, \ldots, A_n \}$ дизъюнктивная форма обобщенного семафорного предиката одной вершины содержит сообщение от другой вершины в каждом слагаемом.*

Доказательство

В основу доказательства положен принцип обязательного последовательного исполнения вершин, использующих критическое данное. Только при этом условии не может возникнуть ситуации, когда эти вершины одновременно получают доступ к одному и тому же данному. Пусть выполняются условия утверждения и обобщенный семафорный предикат вершины A_i содержит сообщение от вершины A_j, тогда в момент времени, когда $R'(A_i)$ принимает истинное значение и управление передается вершине A_i, вершина A_j уже завершилась. Так как эти вершины не могут входить в разные циклы, в течение выполнения оператора вершины A_i вершина A_j не может получить управление. Следовательно, конфликта совместного использования данного *d* не возникает.

Необходимость докажем методом опровержения. Пусть ни один из обобщенных семафорных предикатов вершин A_i и A_j не содержит сообщения от другой вершины. Так как семафорный предикат описывает условие запуска вершины, отсутствие в нем сообщения от определенной вершины означает *независимость* момента времени, в который оператор вершины начинает исполнение, от того, завершилась или нет определенная вершина. Тогда потенциально возможна ситуация, когда интервалы времени исполнения этих двух вершин пересекутся, а значит, возникнет конфликтная ситуация. Следовательно, если при исполнении вершин A_i и A_j не возникает конфликта по данным, то обобщенный семафорный предикат одной из вершин должен содержать сообщение от другой вершины. Необходимость доказана.

Метод проверки корректности синхронизации множества вершин $\widetilde{A} = \{A_{i_1}, \ldots, A_{i_n}\}$, использующих некоторое критическое данное *d*, заключается в построении дизъюнктивной формы обобщенного семафорного предиката для каждой из вершин $A_{i_1}, \ldots, A_{i_n}$ и анализе

содержимого полученных семафорных предикатов. Если существует такая пара вершин $\{A_i, A_j\} \subset \widetilde{A}$, для которой семафорный предикат вершины A_i не содержит сообщения от вершины A_j и при этом семафорный предикат вершины A_j не содержит сообщения от вершины A_i, то конфликт совместного использования данного d вершинами $A_{i_1},...,A_{i_n}$ не разрешен.

5.4.2 Взаимные блокировки в параллельных вычислительных процессах

Введение в модель дуг синхронизации позволяет избавиться от критических данных, но влечет за собой опасность возникновения в модели тупиков.

ОПРЕДЕЛЕНИЕ. ***Тупиком (взаимоблокировкой, дедлоком, клинчем***) будем называть такое состояние вычислительного процесса, при котором он еще не завершен, но ни один из его операторов не может начать исполнение.

Тупики возникают, когда параллельные ветви вычислительного процесса бесконечно долго ожидают появления или освобождения ранее захваченных ресурсов, которые никогда не появятся или не будут освобождены. В рассматриваемой модели параллельных вычислений в качестве таких ресурсов выступают сообщения, пересылаемые между вершинами в соответствии с дугами синхронизации. Если несколько вершин граф-модели будут ожидать сообщения, которые никогда не будут сформированы, возникнет тупик.

Рассмотрим пример (рис. 5.13).

По завершении оператора начальной вершины управление передается вершинам A_1 и A_3, принадлежащим различным параллельным ветвям. Семафорный предикат вершины A_1 содержит сообщение от вершины A_4, а семафорный предикат вершины A_3 – сообщение от вершины A_2. Эти сообщения никогда не поступят в почтовый ящик, так как вершины A_2 и A_4, в которых они формируются, сами получают управление после завершения вершин A_1 и A_3 соответственно. Таким образом, после передачи управления вершинам A_1 и A_3 ни в одной из вершин не сможет начаться исполнение оператора. Возникнет тупик.

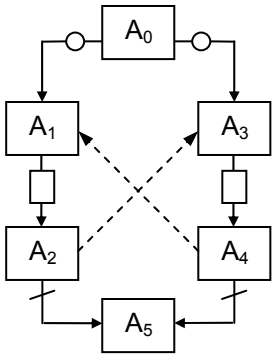


Рис. 5.13. Пример граф-модели, содержащей тупик

Для поиска тупиков в граф-модели параллельного вычислительного процесса воспользуемся введенным ранее понятием состояния модели параллельного вычислительного процесса как множества вершин, в которых происходят вычисления в момент времени t: $S_t = \{ A_{i_0}, \ldots, A_{i_{k-1}} \}$.

Так как в каждой параллельной ветви одновременно может выполняться не более одной вершины, значение k не может превышать числа параллельных ветвей в модели (обозначим его через K) за вычетом мастер-ветви: $0 \le k \le K\text{-}1$.

Построим **покрывающее дерево**, определяющее множество возможных состояний модели параллельного вычислительного процесса. Одноименное понятие используется в теории сетей Петри, где оно определяет множество достижимых в сети разметок и служит для доказательства свойства ограниченности сетей.

Покрывающее дерево будем строить по следующим правилам.

В каждой вершине дерева разместим по одному состоянию модели, которое описывается множеством $S = \{ A_{i_0}, \ldots, A_{i_{k-1}} \}$.

Корневая вершина покрывающего дерева содержит множество $S_0 = \{A_0\}$, где A_0 – начальная вершина модели; S_0 соответствует начальному состоянию модели в момент времени $t = 0$.

Произвольная вершина покрывающего дерева (содержащая состояние S_i) может иметь потомков, описывающих состояния, в кото-

рые способна перейти модель из состояния S_i. Очевидно, что число таких состояний

$$N_{S_i} \le \sum_{j=1}^{k_i} M_j ,$$

где M_j – количество исходящих дуг из вершины A_j; k_i – количество элементов множества S_i.

Так как M_j конечно, а $k_i \le K$-1, где K – конечно, то для модели без циклов покрывающее дерево содержит конечное число вершин.

Отметим, что перечень потомков, которые могут быть добавлены к некоторой вершине A_i, зависит не только от содержимого множества S_i, но и от содержимого вершин, предшествующих вершине A_i, то есть расположенных ближе к корневой вершине покрывающего дерева. Это обусловлено тем, что возможность запуска некоторой вершины модели (т.е. перехода в новое состояние) определяется значением предиката, помечающего дугу, входящую в данную вершину, а также истинностью семафорного предиката этой вершины. При поиске тупиков нас интересуют все возможные пути развития вычислительного процесса, поэтому будем полагать, что переход по любой дуге модели возможен. Для этого достаточно подобрать соответствующий набор входных данных модели. Тогда возможность изменения состояния модели путем перехода по некоторой дуге определятся истинностью семафорного предиката вершины, в которую ведет эта дуга. Семафорный предикат использует сообщения от других вершин модели, отправляемых данной вершине. Наличие этих сообщений при нахождении модели в состоянии S_i зависит от того, какие вершины завершили вычисления на момент перехода модели в состояние S_i, т.е. от предшествующих ему состояний.

В качестве примера на рис. 5.14, б изображено покрывающее дерево, описывающее множество возможных состояний граф-модели, приведенной на рис. 5.14, а.

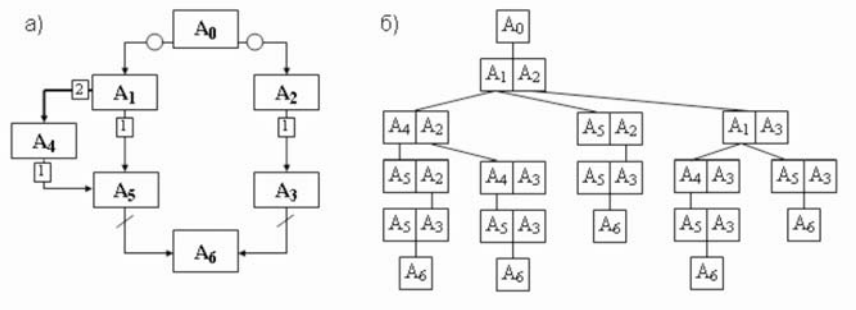


Рис. 5.14. Пример покрывающего дерева, соответствующего граф-модели параллельного вычислительного процесса

Рассмотрим **алгоритм построения покрывающего дерева**, описывающего множество возможных состояний в модели параллельного вычислительного процесса.

В качестве важной компоненты алгоритма построения покрывающего дерева опишем правила добавления потомков для вершины покрывающего дерева, содержащей состояние S_i.

Для $\forall A_j \in S_i$, $j = 0, \ldots, k_i$, строится множество $W_j = \{A_{j0}, \ldots, A_{j\psi_j}\}$ вершин, в которые ведут дуги из вершины A_j. Если для некоторой вершины $A_k \in W_j$ семафорный предикат $R_{Ak} = 1$, то к вершине покрывающего дерева S_i добавляется потомок, множество S_k которого получается из S_i:

1) заменой A_j на A_k, если $T(\Psi_{jk}) = 1$, т.е. в A_k ведет последовательная дуга;

2) заменой всех $A_j \mid \Psi_{jk} \in \Psi_G$ единственной вершиной A_k, если $T(\Psi_{jk}) = 3$ (в A_k ведет терминирующая дуга);

3) заменой A_j на множество $\{A_{j0}, \ldots, A_{jn}\} \mid (T(\Psi_{j,jm}) = 2) \wedge (R_{Ajm} = 1)$, $m = 0, \ldots, n$, если $T(\Psi_{jk}) = 2$ (в A_k ведет параллельная дуга).

Добавление потомка к вершине дерева становится невозможным:

1) если текущая вершина i содержит $S_i = \{A_N\}$, где A_N – конечная вершина граф-модели, из которой не исходит ни одной дуги;

2) для $\forall A_j \in S_i : R_{Ak} = 0$, $k = j0, \ldots, j\psi_j$, т.е. ни одна из вершин, в которые ведут дуги из вершин текущего состояния S_i, не может начать исполнение. Такая ситуация определяет **тупик**.

В соответствии с описанными выше правилами добавления потомков для произвольной вершины, содержащей состояние S_i, можно предложить следующий алгоритм построения покрывающего дерева:

1. Воспользуемся следующими переменными:

int A – номер вершины граф-модели;

pCoverTree CT – указатель на элемент структуры, описывающей покрывающее дерево (описание структуры см. ниже);

pLpost Lpost – указатель на список сообщений, пересылаемых между вершинами граф-модели;

pConRA ConRA – указатель на список сообщений, входящих в семафорный предикат вершины граф-модели с номером A;

tTopList W – указатель на список номеров вершин граф-модели;

tTopList Si – указатель на список номеров вершин, описывающий состояние граф-модели;

tTopList Si1 – состояние граф-модели (переменная для хранения промежуточных значений);

tTopList Sr – состояние граф-модели (переменная для хранения промежуточных значений).

2. Пронумеруем все вершины граф-модели параллельного вычислительного процесса. В вершинах дерева будем хранить множества $S_i = \{s_0, \ldots, s_k\}$, где $s_0, \ldots, s_k$ – номера вершин граф-модели, в которых происходят вычисления при нахождении модели в состоянии S_i. Кроме того, каждая вершина дерева содержит множество $V_i = \{v_0, \ldots, v_k\}$, где $v_j \in \{0,1\}$, $j = 0, \ldots, k$ – флаг проверки, показывающий, исследована ли алгоритмом возможность изменения состояния граф-модели в результате завершения вычислений в вершине с номером s_j. В памяти ЭВМ покрывающее дерево представляется структурой, изображенной на рис. 5.15.

3. В корневую вершину дерева записать начальную вершину граф-модели: $S_0 = \{s_0\}$, $V_0 = \{0\}$.

4. Найти любую вершину с состоянием S_i, еще не объявленную листом (то есть вершиной, не имеющей потомков). Если таких вершин нет, перейти к п. 5. Если вершина еще не объявлена листом: $A = S_j \mid v_j \neq 1$. Если $\forall j = 0, \ldots, k_i,\ v_j = 1$, то вершину S_i объявить листом, перейти к п. 4.

4.1) $v_j = 1$;

4.2) построить W – список номеров вершин граф-модели, в которые ведут дуги из вершины с номером A:

$W = \{w_0, \ldots, w_l\} \mid \psi_{A,wj} \in \Psi_G$. Если $W = \varnothing$, перейти к п. 4;

4.3) в список сообщений L_{post} добавить сообщения от вершины с номером A: $L_{post} = L_{post} \cup \{\mu_{A,j}\}$, $j = j0, \ldots, j\psi_A$. Из списка сообщений L_{post} удалить ConRA – сообщения, входящие в семафорный предикат R_A;

4.4) вычислить семафорные предикаты R_j для всех вершин w_j, $j = 0, \ldots, l$. $S_r = S \setminus [A]$. $j = 0$. AddPar = 0;

4.5) пока $j \le l$:

4.5.1) Если $R_j = 1$, то перейти к п. 4.5.2), иначе $j = j + 1$, перейти к п. 4.5,

4.5.2) если $T(\Psi_{A,j}) = 2$, то $S_r = S_r \cup \{w_j\}$, AddPar = 1, $j = j + 1$, перейти к п. 4.5,

4.5.3) если $T(\Psi_{A,j}) = 3$, то $S_r = (S_r \setminus S_T) \cup \{w_j\}$, где $S_T = \{s_{T1}, \ldots, s_{Tn}\}$, $\Psi_{Tr,j} \in \Psi_G$, $r = 1, \ldots, n$,

4.5.4) если $T(\Psi_{A,j}) = 1$, то $S_r = S_r \cup \{w_j\}$,

4.5.5) добавить потомка к вершине S_i:

$S_{i1} = S_r$, $V_{i1} = \{v_0, \ldots, v_{k1}\}$, $v_r = 0$, $r = 0, \ldots, k1$,

$j = j + 1$, перейти к п. 4.5;

4.6) если AddPar = 1, добавить потомка к вершине S_i:

$S_{i1} = S_r$, $V_{i1} = \{v_0, \ldots, v_{k1}\}$, $v_r = 0$, $r = 0, \ldots, k1$.

5. Перейти к п. 4.

6. Конец работы.

Используя понятие покрывающего дерева, можно предложить следующий метод нахождения тупиков в модели параллельного вычислительного процесса:

1. Строится покрывающее дерево для данной модели параллельного вычислительного процесса.

2. Если покрывающее дерево содержит вершину без потомков (лист), множество S_i которой не равно $\{A_N\}$, где A_N – конечная вершина граф-модели, это означает, что из состояния S_i невозможен переход в другое состояние, т.е. модель содержит тупик. Вершины, в которых остановится вычислительный процесс при возникновении этого тупика, определяются значением множества S_i.

3. Если все вершины покрывающего дерева, не имеющие потомков, содержат множество $S_i = \{A_N\}$, где A_N – конечная вершина граф-

модели, то в модели не может возникнуть состояния, из которого не возможен переход в другое состояние, т.е. в модели нет тупиков.

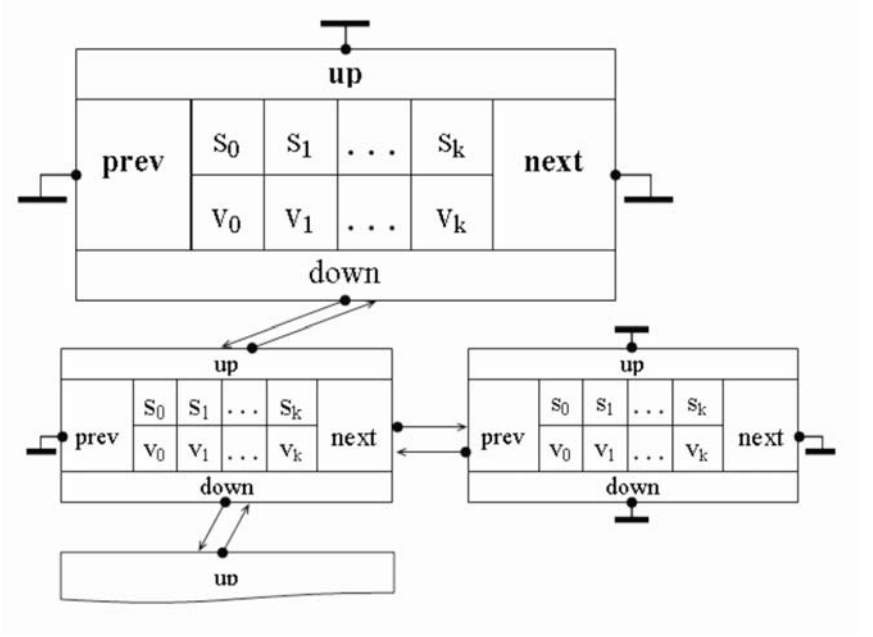


Рис. 5.15. Представление покрывающего дерева в памяти ЭВМ

Рассмотрим пример простейшей модели параллельных вычислений, изображенной на рис. 5.16.

Предположим, что вершины A_2 и A_5 используют для записи одно и то же данное. Для того чтобы исключить конфликт по данным, необходимо соединить эти вершины дугой синхронизации. Заметим, что если провести дугу синхронизации из вершины A_5 в вершину A_2, то мы получим последовательную программу, так как правая ветвь не начнет исполнение до завершения последней вершины левой ветви. Более целесообразно провести дугу синхронизации из вершины A2 в вершину A5. На рис. 5.17 изображена модель с дугой синхронизации (рис. 5.17, а) и соответствующее этой модели покрывающее дерево (рис. 5.17, б).

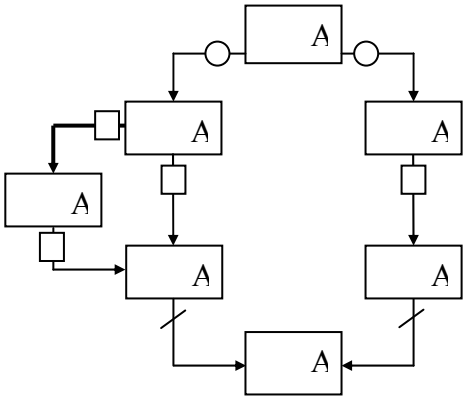


Рис. 5.16. Пример модели параллельных вычислений

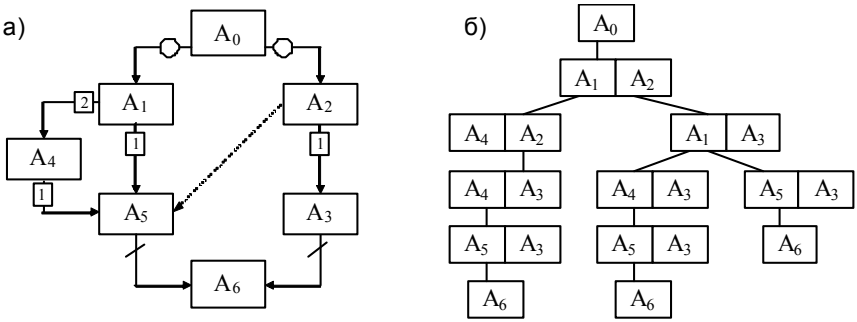


Рис. 5.17. Синхронизация вершин, совместно использующих данные

Из рисунка видно, что во всех вершинах покрывающего дерева, не имеющих потомков, содержится конечная вершина граф-модели. Это означает отсутствие тупиков в модели вычислительного процесса.

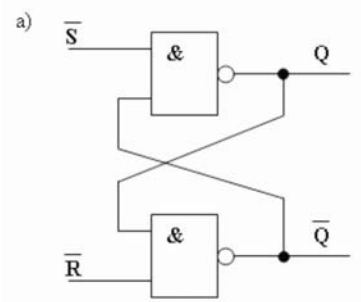
5.5. Пример использования методов поиска критических данных и проверки корректности синхронизации

5.5.1 Параллельная модель RS-триггера

Проиллюстрируем применение рассмотренных методов на реальном примере. Для этого создадим модель асинхронного потенциального RS-триггера, построенного на элементах И-НЕ. Принципиальная схема такого триггера изображена на рис. 5.18, а. Таблица истинности логического функционирования RS-триггера приведена на рис. 5.18, б.

В литературе известно несколько описаний модели RS-триггера: модель блок-схем, сеть Петри, автоматная и рекурсивная модели. Мы смоделируем работу триггера, используя технологию ГСП.

Следует отметить, что, рассматривая различные аспекты работы триггера, можно построить несколько его моделей (физическую, электрическую, логическую, тепловую и т.д.). Например, электрическая модель рассматривает триггер как электрическую цепь, состоящую из различных элементов (резисторов, транзисторов и т.п.), функционирование которой описывается законами электротехники. Мы создадим логическую модель, отражающую функционирование триггера как дискретного логического элемента, выдающего на выходах Q и $\overline{Q}$ двоичные значения в зависимости от значений на входах элемента $\overline{R}$ и $\overline{S}$.



б)

$\overline{R}$	$\overline{S}$	Q	$\overline{Q}$	Описание режима работы
0	0	1	1	Запрещено
0	1	1	0	Установка Q в 1
1	0	0	1	Установка Q в 0
1	1	Q	$\overline{Q}$	Хранение (значения выходов зависят от предыдущего состояния)

Рис. 5.18. Принципиальная схема RS-триггера на элементах И-НЕ

Логика работы RS-триггера во времени может быть описана следующей системой функций (переменные x11 и x22 описаны в табл. 5.1):

$$\begin{cases} y1(t+1) = \overline{x11(t) \,\&\, y2(t)} \\ y2(t+1) = \overline{x22(t) \,\&\, y1(t)} \end{cases}. \qquad (5.8)$$

Поскольку триггеры в основном применяются в цифровых устройствах, они, как правило, работают в дискретном времени и значения их выходных сигналов анализируются только в определенные моменты времени. Переменная t в системе (5.8) изменяет свои значения дискретно. Выражение (5.8) показывает, что значения выходов триггера на следующем шаге дискретного времени определяются не только значениями входных сигналов $\overline{\mathrm{R}}$ и $\overline{\mathrm{S}}$, но и значениями его выходов на текущем шаге. Таким образом, работу триггера следует рассматривать в динамике.

Для построения модели триггера воспользуемся несколькими переменными, описывающими значения двоичных сигналов на входах и выходах триггера. Активный уровень сигнала обозначим 1, пассивный – 0. Список переменных приведен в табл. 5.1.

Таблица 5.1. Список используемых переменных в логической модели RS-триггера

Название переменной	Описание
R	Значение сигнала на внешнем входе $\overline{R}$
S	Значение сигнала на внешнем входе $\overline{S}$
x11	Значение сигнала на верхнем входе верхнего элемента И-НЕ
x12	Значение сигнала на нижнем входе верхнего элемента И-НЕ
x21	Значение сигнала на верхнем входе нижнего элемента И-НЕ
x22	Значение сигнала на нижнем входе нижнего элемента И-НЕ
y1	Значение сигнала на выходе верхнего элемента И-НЕ (выход Q)
y2	Значение сигнала на выходе верхнего элемента И-НЕ (выход $\overline{Q}$)

Модель верхнего логического элемента И-НЕ приведена на рис. 5.19.

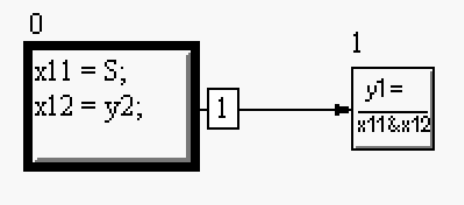


Рис. 5.19. Модель логического элемента И-НЕ

Граф-модель элемента И-НЕ состоит из двух вершин. В корневой вершине считываются значения входных сигналов, а во второй вершине вычисляется логическое значение выходного сигнала. На рис. 5.19 изображена модель верхнего элемента И-НЕ для схемы триггера, показанной на рис. 5.18. Модель нижнего элемента И-НЕ отличается только используемыми переменными (x21, x22, R, y1, y2). Модель триггера приведена на рис. 5.20.

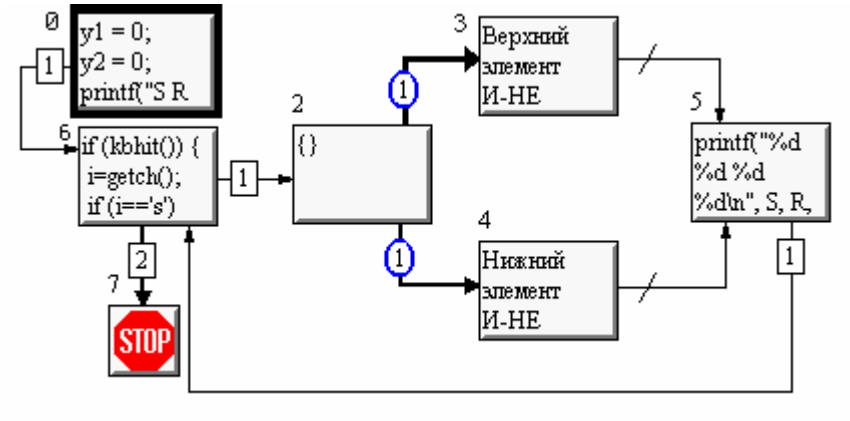


Рис. 5.20. Модель асинхронного RS-триггера

В вершине 0 происходит инициализация переменных, после чего модель выполняет цикл, состоящий из вершин *6, 2, 3* и *4, 5* и повторяющийся до получения команды о завершении моделирования. Эта команда формируется нажатием клавиши «2» на клавиатуре.

Изменение входных сигналов моделируется нажатием различных клавиш. Анализ нажатий выполняется в вершине 6, которая присваивает соответствующие значения входным сигналам $\overline{R}$ и $\overline{S}$ (переменные x22 и x11 соответственно). Нажатие клавиши «S» присваивает «1» переменной x11, нажатие клавиши «R» присваивает «1» переменной x22. Клавиши «X» и «F» присваивают «0» соответственно переменным «S» и «R». Клавиша «1» одновременно присваивает «1» обеим переменным x11 и x22, клавиша «0» одновременно присваивает этим переменным «0».

Вершина *2* запускает две параллельные ветви, каждая из которых моделирует работу одного из элементов И-НЕ и формирует соответствующий выходной сигнал. Верхняя ветвь на рис. 5.20 (вершина *3*) моделирует работу верхнего элемента И-НЕ триггера (см. схему на рис. 5.18, а) и формирует выходной сигнал Q. Нижняя ветвь (вершина *4*) моделирует нижний элемент И-НЕ и формирует сигнал $\overline{Q}$. Полученные результаты визуализируются в вершине *5*, после чего управление возвращается в вершину *6*.

Каждый прогон цикла моделирует работу триггера на одном шаге дискретного времени. Полученные на этом шаге значения выходных сигналов сохраняются в переменных y1 и y2 и используются как входные сигналы (переменные) на следующем шаге (см. обратные связи на рис. 5.18, а). Следует отметить, что сигнал на выходе элемента И-НЕ формируется с некоторой задержкой, поэтому можно полагать, что момент считывания значений входных сигналов и момент формирования выходных сигналов разнесены во времени. Поскольку верхний элемент И-НЕ использует выходной сигнал нижнего элемента в качестве входного (и наоборот), модель должна гарантировать, что на каждом шаге дискретного времени на входе каждого из элементов будет значение выходного сигнала другого элемента, полученное на предыдущем шаге. Покажем, что обеспечение этого требования означает корректное использование совместных данных y1 и y2, одновременно изменяющихся в вершинах *3* и *4*. Для этого изобразим модель триггера, не содержащую иерархии (рис. 5.21).

Пунктирной линией на рис. 5.21 обведены модели элементов И-НЕ триггера, лежащие в параллельных вершинах *3* и *4* графа на рис. 5.20.

Из рис. 5.21 видно, что параллельные ветви совместно используют переменные y1 и y2. Вершина 5.0, принадлежащая верхней параллельной ветви, использует переменную y2 для чтения, а вершина 4.1 нижней ветви использует ту же переменную для записи. Аналогично переменная y1 используется вершинами 4.0 и 3.1.

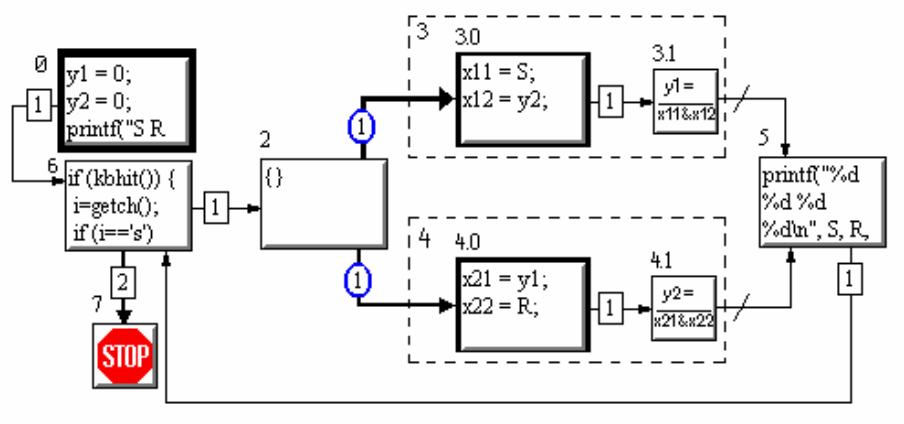


Рис. 5.21. Граф-модели триггера без иерархии

Построим формулу над способами использования данных в модели на рис. 5.21, заменив в ней обозначения способа использования данного в каждой вершине на номер этой вершины ($H(A_i,d) \rightarrow i$):

$$H(G,d) = 0 \Delta (6 \Delta (7 \nabla (\ 2 \Delta [(3.0 \Delta 3.1) \# (4.0 \Delta 4.1)] \Delta 5))). \quad (5.9)$$

Способы использования данных в каждой вершине и в модели в целом приведены в табл. 5.2.

Из таблицы видно, что данные y1 и y2 являются критическими, то есть возможны ошибки их совместного использования параллельными ветвями модели. Рассмотрим переменную y1. Действительно, если не вводить в модель синхронизацию, то момент считывания значения переменной y1 в вершине 4.0 и момент присвоения ей значения в вершине 3.1 будут несогласованными. В результате при повторении цикла некоторые значения переменной y1 могут быть пропущены. Если на текущей итерации вершина 4.0 прочитала значение y1 до того, как оно изменилось в вершине 3.1, а на следующей итера-

ции вершина 3.1 выполнилась раньше вершины 4.0, то значение y1, вычисленное на текущей итерации, будет потеряно. Применительно к работе триггера такая ситуация означает, что нижний элемент И-НЕ (по схеме на рис. 5.18, а) обработает сигнал Q, полученный на текущем шаге дискретного времени, а не на предыдущем. Таким образом, наличие критических данных приводит к некорректной работе модели.

Таблица 5.2. Способы использования данных в логической модели RS-триггера

Имя данного	**Способ использования данного в вершине:**																		
	0		**6**		**7**		**2**		**3.0**		**3.1**		**4.0**		**4.1**		**5**		**H(G|d)**
R	2	Δ(	2	Δ(	1	∇(	0	Δ[(	0	Δ	0	)#(	1	Δ	0	)]Δ	1	))=	2
S	2	Δ(	2	Δ(	1	∇(	0	Δ[(	1	Δ	0	)#(	0	Δ	0	)]Δ	1	))=	2
x11	2	Δ(	0	Δ(	0	∇(	0	Δ[(	2	Δ	1	)#(	0	Δ	0	)]Δ	0	))=	2
x12	2	Δ(	0	Δ(	0	∇(	0	Δ[(	2	Δ	1	)#(	0	Δ	0	)]Δ	0	))=	2
x21	2	Δ(	0	Δ(	0	∇(	0	Δ[(	0	Δ	0	)#(	2	Δ	1	)]Δ	0	))=	2
x22	2	Δ(	0	Δ(	0	∇(	0	Δ[(	0	Δ	0	)#(	2	Δ	1	)]Δ	0	))=	2
y1	2	Δ(	0	Δ(	0	∇(	0	Δ[(	0	Δ	2	)#(	1	Δ	0	)]Δ	1	))=	**3**
y2	2	Δ(	0	Δ(	0	∇(	0	Δ[(	1	Δ	0	)#(	0	Δ	2	)]Δ	1	))=	**3**

Введение в модель синхронизации должно учитывать логику работы триггера. Необходимо, чтобы выходные сигналы элементов И-НЕ на текущем шаге дискретного времени формировались на основе значений этих сигналов, полученных на предыдущем шаге, и только после того, как эти значения поступят на вход элементов И-НЕ. Синхронизированная модель, построенная с учетом этого правила, изображена на рис. 5.22.

Введенная синхронизация сохраняет возможность одновременного исполнения пар вершин 3.0 и 4.0, 3.1 и 4.1 и в то же время гарантирует, что присвоение нового значения переменным y1 и y2 про-

изойдет после того, как текущее значение будет прочитано в вершинах 3.0 и 4.0.

Построим обобщенные предикаты вершин 3.0, 3.1, 4.0 и 4.1 и убедимся, что введенная синхронизация исключает конфликт совместного использования данных y1 и y2.

$R'(A_{3.0}) = R'(A_2) \wedge b_{2,3.0} = (R'(A_6) \wedge b_{6,2}) \wedge b_{2,3.0} = ((1 \wedge b_{0,6}) \wedge b_{6,2}) \wedge b_{2,3.0}$,

$R'(A_{4.0}) = R'(A_2) \wedge b_{2,4.0} = (R'(A_6) \wedge b_{6,2}) \wedge b_{2,4.0} = ((1 \wedge b_{0,6}) \wedge b_{6,2}) \wedge b_{2,4.0}$,

$R'(A_{3.1}) = R'(A_{3.0}) \wedge b_{3.0,3.1} \wedge R'(A_{4.0}) \wedge b_{4.0,3.1} =$

$= 1 \wedge b_{0,6} \wedge b_{6,2} \wedge b_{2,3.0} \wedge b_{3.0,3.1} \wedge 1 \wedge b_{0,6} \wedge b_{6,2} \wedge b_{2,4.0} \wedge b_{4.0,3.1} =$

$= 1 \wedge b_{0,6} \wedge b_{6,2} \wedge b_{2,3.0} \wedge b_{3.0,3.1} \wedge b_{2,4.0} \wedge b_{4.0,3.1}$,

$R'(A_{4.1}) = R'(A_{4.0}) \wedge b_{4.0,4.1} \wedge R'(A_{3.0}) \wedge b_{3.0,4.1} =$

$= 1 \wedge b_{0,6} \wedge b_{6,2} \wedge b_{2,4.0} \wedge b_{4.0,4.1} \wedge b_{2,3.0} \wedge b_{3.0,4.1}$. (5.10)

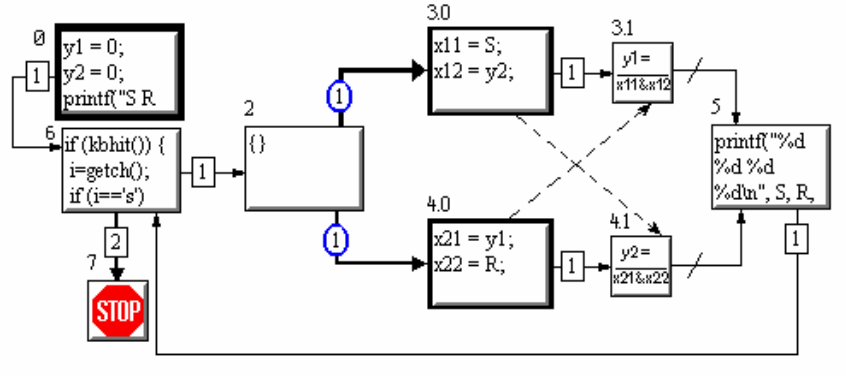


Рис. 5.22. Модель триггера с синхронизацией параллельных ветвей

Переменные y1 и y2 используются парами вершин 3.0, 4.1 и 4.0, 3.1. В паре вершин 3.0, 4.1 обобщенный предикат одной из вершин ($R'(A_{4.1})$) содержит сообщение от другой вершины ($b_{3.0,4.1}$). Аналогично в паре 4.0, 3.1 обобщенный предикат $R'(A_{3.1})$ содержит сообщение $b_{4.0,3.1}$. Согласно утверждению 3, конфликт совместного использования данных в вершинах 3.0, 4.1 и 4.0, 3.1 отсутствует.

Проверим теперь, что введенная синхронизация не приводит к возникновению тупиков. Для этого построим покрывающее дерево для графа, изображенного на рис. 5.22. Поскольку тупик может возникнуть только на параллельном участке модели, возврат управления в цикле от вершины *5* в вершину *6* не влияет на возникновение тупиков. Действительно, в каждой итерации цикла параллельный фрагмент повторяется целиком, и возможность запуска любой из параллельных вершин не зависит от предыдущих итераций. Тогда для обнаружения возможных тупиков можно рассматривать одну итерацию цикла, исключив из модели дугу от вершины *5* к вершине *6*. Вершина *5* становится в таком случае конечной вершиной. Покрывающее дерево для модели триггера, содержащей один прогон цикла, изображено на рис. 5.23.

Из рис. 5.23 видно, что покрывающее дерево содержит два листа, в каждом из которых содержится конечная вершина модели. Следовательно, введенная синхронизация не приводит к возникновению тупиков.

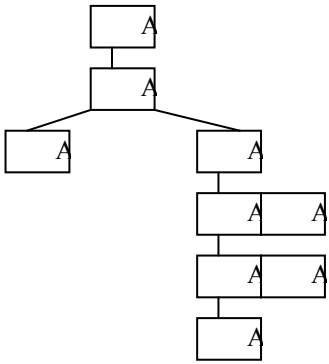


Рис. 5.23. Покрывающее дерево для модели RS-триггера

5.5.2 Модель RS-триггера без синхронизации

Как упоминалось выше, кроме введения в модель дуг синхронизации критические данные можно устранить переименованием переменных. Построим модель RS-триггера, не содержащую критических данных.

Введем дополнительные переменные y11 и y22, в которых будем сохранять значения сигналов y1 и y2, полученные на предыдущем шаге дискретного времени. На вход элементов И-НЕ будем подавать новые переменные вместо переменных y1 и y2, тем самым исключив их одновременное использование параллельными ветвями для чтения и записи.

Модель триггера примет вид, изображенный на рис. 5.24.

В вершинах 3.0 и 4.0 вместо переменных y1 и y2 используются для чтения переменные y11 и y22, которые получают новые значения перед началом параллельного фрагмента модели (вершина *2*). В результате пары вершин 3.0, 4.1 и 4.0, 3.1 используют для чтения и записи различные переменные.

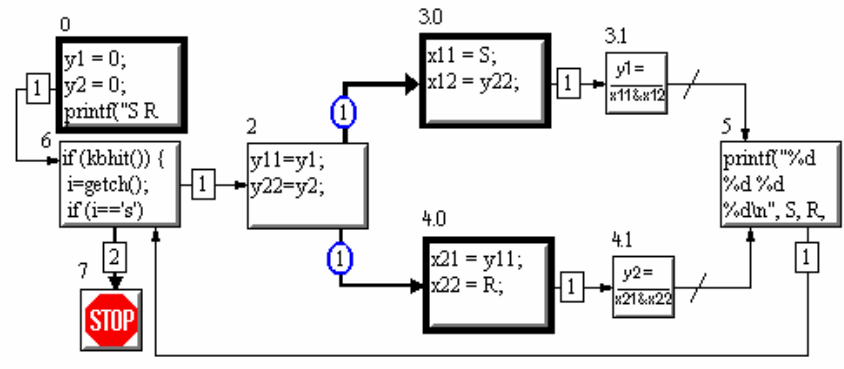


Рис. 5.24. Модель RS-триггера без синхронизации

Построим формулы над способами использования данных и вычислим способ использования каждой из переменных (табл. 5.3).

Из табл. 5.3 видно, что способ использования всех данных в модели триггера, полученной путем переименования переменных, равен 2, поэтому критические данные в ней отсутствуют.

Преимуществом модели, изображенной на рис. 5.24, является отсутствие дуг синхронизации, а значит, возможности возникновения тупиковых ситуаций. Недостатком модели является большее количество переменных. Хотя в данной модели это увеличение практически не заметно, в относительных величинах оно составляет 25%. Отсюда

можно сделать вывод, что для более сложных моделей переименование переменных может привести к значительному усложнению их анализа. По этой причине рекомендуется применять оба способа исключения ошибок совместного использования данных: введение синхронизации в модель и переименование переменных.

Таблица 5.3. Способы использования данных в логической модели RS-триггера без синхронизации

Имя данного	Способ использования данного в вершине																		
	0		**6**		**7**		**2**		**3.0**		**3.1**		**4.0**		**4.1**		**5**		**H(G|d)**
R	2	Δ(	2	Δ(	1	∇(	0	Δ[(	0	Δ	0	)#(	1	Δ	0	)]Δ	1	)))=	**2**
S	2	Δ(	2	Δ(	1	∇(	0	Δ[(	1	Δ	0	)#(	0	Δ	0	)]Δ	1	)))=	**2**
x11	2	Δ(	0	Δ(	0	∇(	0	Δ[(	2	Δ	1	)#(	0	Δ	0	)]Δ	0	)))=	**2**
x12	2	Δ(	0	Δ(	0	∇(	0	Δ[(	2	Δ	1	)#(	0	Δ	0	)]Δ	0	)))=	**2**
x21	2	Δ(	0	Δ(	0	∇(	0	Δ[(	0	Δ	0	)#(	2	Δ	1	)]Δ	0	)))=	**2**
x22	2	Δ(	0	Δ(	0	∇(	0	Δ[(	0	Δ	0	)#(	2	Δ	1	)]Δ	0	)))=	**2**
y11	2	Δ(	0	Δ(	0	∇(	2	Δ[(	0	Δ	0	)#(	1	Δ	0	)]Δ	1	)))=	**2**
y22	2	Δ(	0	Δ(	0	∇(	2	Δ[(	1	Δ	0	)#(	0	Δ	0	)]Δ	1	)))=	**2**
y1	2	Δ(	0	Δ(	0	∇(	1	Δ[(	0	Δ	2	)#(	0	Δ	0	)]Δ	1	)))=	**2**
y2	2	Δ(	0	Δ(	0	∇(	1	Δ[(	0	Δ	0	)#(	0	Δ	2	)]Δ	1	)))=	**2**

5.6 Краткий обзор

Использование визуальных средств моделирования параллельных вычислений существенно упрощает разработку параллельных программ в технологическом смысле. Однако процессы составления модели параллельных вычислений и разработка правильно работающей программы существенно отличаются друг от друга.

Дело в том, что моделирование параллельных вычислений требует от разработчика более высокого уровня абстрактного мышления, когда работу алгоритма необходимо представлять не в виде жестко заданной последовательности операций, а во времени, когда процессы, взаимодействуя между собой по сложным схемам, изменяют данные программы. В общем случае развитие сценариев взаимодей-

ствия параллельных процессов носит комбинаторный характер и разработчику чрезвычайно сложно угадать все последствия принятой схемы параллельных вычислений в конечном результате. Кроме того, необходимо помнить, что организация параллельных вычислений производится с единственной целью – ускорить вычисления. А ускорение вычислений как раз не всегда достигается, чего не скажешь про возникновение большого количества ошибочных ситуаций.

В технологии ГСП используются методы автоматического поиска ошибочных ситуаций, результаты работы которых предоставляются разработчику для внесения соответствующих корректировок в разрабатываемый алгоритм.

К методам контроля корректности относятся:

- алгоритм поиска критических данных, основанный на алгебре использования данных;
- метод проверки корректности синхронизации граф-модели;
- метод поиска тупиковых ситуаций.

5.7 Контрольные вопросы

1. Какие данные называются критическими?
2. Дайте определение понятию состояние модели параллельных вычислений.
3. Опишите простейший алгоритм поиска критических данных.
4. Какие операции используются в алгебре способов использования данных?
5. Чем отличаются друг от друга операции следования и ветвления?
6. Опишите правило построения формул использования данных.
7. Свойства операций следования, ветвления и параллельного исполнения.
8. Применение формул над способами использования данных.
9. Что такое семафорный предикат?
10. Дайте определение обобщенного семафорного предиката.
11. Когда критическое данное не приводит к конфликту?
12. В каких случаях в параллельных вычислениях возникают тупиковые ситуации?
13. Что такое покрывающее дерево?
14. Как распознать тупиковые ситуации?

ЛИТЕРАТУРА

Основная литература

1. Воеводин, В.В. Модели и методы в параллельных процессах [Текст] / В.В. Воеводин. – М.: Наука, 1987.
2. Воеводин, В.В. Параллельные вычисления [Текст] / В.В. Воеводин, Вл.В. Воеводин - СПб.: БХВ-Петербург, 2002.
3. Гергель, В.П. Теория и практика параллельных вычислений [Текст] / В.П. Гергель – М.: Интернет-Университет Информационных Технологий; БИНОМ, 2007.
4. Коварцев, А.Н. Автоматизация разработки и тестирования программных средств [Текст] / А.Н. Коварцев. – Самара: СГАУ, 1999.
5. Шалыто, А.А. SWITCH-технология. Алгоритмизация и программирование задач логического управления [Текст] / А.А. Шалыто – СПб.: Наука, 1998.

Дополнительная литература

6. G. Hutton Programming in Haskell [Text] – Cambridge University Press, 2007.
7. Martin, J. Application development without programmers [Text] / J. Martin, R. Murch. – In Savant Inst. seminar documentation by J. Martin. Carnfoth: Savant Res. Studies.
8. Абрамов, С.М. Кроссплатформенная версия Т-системы с открытой архитектурой [Текст]/ С.М. Абрамов, А.А. Кузнецов, В.А. Роганов // Вычислительные методы и программирование. – 2007. – Т. 8, разд. 2. – С. 18-23.
9. Бетелин, В.Б. Системы автоматизации труда программиста [Текст] / В.Б. Бетелин. – М.: Наука, 1990.
10. Братко, И. Программирование на языке ПРОЛОГ для искусственного интеллекта [Текст]/ И. Братко. – М.: Мир, 1990.
11. Вельбицкий, И.В. Технология программирования [Текст]/ И.В. Вельбицкий. – Киев: Техника, 1984.
12. Вирт, Н. Алгоритмы и структуры данных [Текст] / Н. Вирт. – М.: Мир, 1989.

13. Дьяконов, В.П. MATLAB 6/6.1/6/5 + Simulink 4/5 в математике и моделировании [Текст]: полное руководство пользователя / В.П. Дьяконов. – М.: СОЛОН-Пресс, 2003.
14. Замулин, А.В. Системы программирования баз данных и знаний [Текст] / А.В. Замулин. – Новосибирск: Наука, 1990.
15. Липаев, В.В. Отладка сложных программ. Методы, средства, технология [Текст] / В.В. Липаев. – М.: Энергоатомиздат, 1993.
16. Мейер, Б. Методы программирования [Текст]: В 2 т. Т.2. / Б. Мейер, К. Бодуэн – М.: Мир, 1982.
17. Т-система с открытой архитектурой [Текст]/ С.М. Абрамов, А.И. Адамович, А.В. Инюхин [и др.] // Тр. междунар. науч. конф. "Суперкомпьютерные системы и их применение. SSA'2004", 26-28 октября 2004 г. – Минск: ОИПИ НАН Беларуси.
18. Успенский, В.А. Теория алгоритмов: основные открытия и приложения [Текст] / В.А. Успенский, А.Л. Семенов. – М.: Наука, 1987.
19. Евстигнеев, В.А. Применение теории графов в программировании [Текст] / В.А. Евстигнеев. – М.: Наука, 1985.
20. Шалыто, А.А. Ханойские башни и автоматы. [Текст] / А.А. Шалыто, Н.И. Туккель // Программист. – 2002. – №8. – С. 82-90.
21. Евтушенко, Ю.Г. Метод половинного деления для глобальной оптимизации функции многих переменных [Текст] / Ю.Г. Евтушенко, В.А. Ратькин. // Техническая кибернетика. – 1987. – №1. – С. 119-127.
22. Коварцев, А.Н. К вопросу об эффективности параллельных алгоритмов глобальной оптимизации функций многих переменных [Текст] / А.Н. Коварцев, Д.А. Попова-Коварцева // Компьютерная оптика. – 2011. – Т. 35. – № 2. – С. 256 – 262.

Информационные ресурсы сети ИНТЕРНЕТ

23. http:/nowostey.net/software/19826-labview-90.html
24. www.insat.ru/products/?category=8
25. www.netlib.org/hence/hence-2.0-doc-html/hence-2.0-doc.html
26. www.softcraft.ru/parallel/fpp/fppcontent.shtml

Учебное издание

Коварцев Александр Николаевич
Жидченко Виктор Викторович

МЕТОДЫ И СРЕДСТВА ВИЗУАЛЬНОГО ПАРАЛЛЕЛЬНОГО ПРОГРАММИРОВАНИЯ. АВТОМАТИЗАЦИЯ ПРОГРАММИРОВАНИЯ

Учебник

Редактор Т.К. Кретинина
Доверстка И.И. Спиридонова

Подписано в печать 15.11.11 г.
Формат 60x84 1/16. Бумага офсетная. Печать офсетная.
Печ. л. 10,5. Тираж 20 экз. Заказ . Арт. 1(Д2)/2011

Самарский государственный аэрокосмический университет
443086, г. Самара, Московское шоссе, 34.

Издательство Самарского государственного
аэрокосмического университета
443086, г. Самара, Московское шоссе, 34.