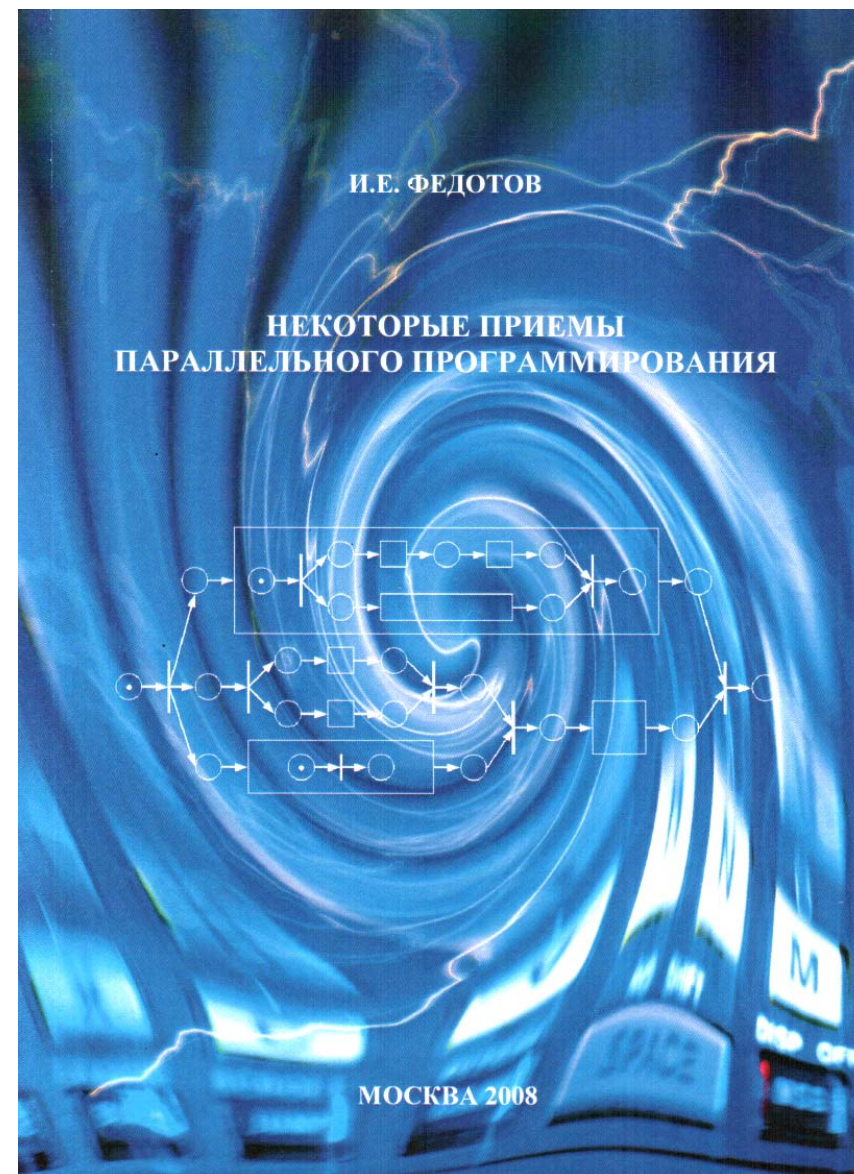


| | |
|--|------------|
| 3.3.2 Простая реализация с использованием MPI..... | 92 |
| 3.3.3 Реализация с поддержкой вложенных сетей..... | 95 |
| 3.4 Бильярдные шары | 103 |
| 3.5 Сумматор | 112 |
| Глава 4. Сети Петри..... | 121 |
| 4.1 Краткое введение в теорию сетей Петри | 121 |
| 4.1.1 Знакомство с сетями Петри | 121 |
| 4.1.2 Строго иерархические сети | 125 |
| 4.1.3 Параллельные вычисления и синхронизация | 127 |
| 4.1.4 Задача об обедающих философах | 130 |
| 4.2 Пример реализации механизма сетей Петри | 133 |
| 4.2.1 Функционирование строго иерархических сетей | 134 |
| 4.2.2 Выполнение параллельных процессов | 145 |
| Заключение..... | 157 |
| Приложение 1. Шаблоны классов матрицы и вектора | 159 |
| Приложение 2. Классы построения и выполнения комплекса работ..... | 163 |
| Приложение 3. Классы построения и выполнения сетей конечных автоматов | 167 |
| Приложение 4. Классы построения и выполнения сетей Петри..... | 174 |
| Библиографический список | 182 |
| Оглавление..... | 186 |



Глава 4. Сети Петри

Текущая глава посвящена вопросу создания параллельных программ на основе сетей Петри. Сети Петри представляют собой аппарат моделирования динамических дискретных систем и являются одним из наиболее адекватных способов описания асинхронного выполнения параллельных процессов, в том числе в распределенных системах.

Следует отметить, что, поскольку сети Петри изначально предназначены больше для моделирования систем, представление на их базе архитектуры проектируемой программной системы, в том числе параллельной, а также последующая ее реализация, для некоторых разработчиков оказываются сопряженными с некоторыми сложностями. Это является следствием не слишком высокой согласованности этой модели с популярными на текущий момент методами программирования. Мы же здесь покажем, что, несмотря на это, они являются более мощным средством построения параллельных вычислений, включающим в себя возможности существующих популярных средств. Кроме того, представление вычислительного процесса в виде сети Петри может быть не привязано к самой реализации ее функционирования и может быть построено путем выделения подзадач в отдельные функциональные блоки. На совести сети Петри в этой ситуации остается лишь организация последовательности их выполнения и синхронизация.

4.1 Краткое введение в теорию сетей Петри

Мы не будем сильно углубляться в описание теории сетей Петри, поскольку это довольно широкая тема, а ограничимся лишь поверхностным описанием, достаточным для иллюстрации их использования при построении параллельных программ. Для более детального ознакомления следует обратиться к специально посвященной этой теме литературе [11, 15, 21].

4.1.1 Знакомство с сетями Петри

В некотором приближении можно сказать, что сеть Петри обобщает понятие конечного автомата и добавляет ему некото-

Оглавление

| | |
|--|-----------|
| Введение..... | 3 |
| О целях издания..... | 3 |
| О проблеме параллельного программирования | 5 |
| Об используемой терминологии | 8 |
| Глава 1. Интерфейсы и технологии параллельного программирования | 10 |
| 1.1 Интерфейс OpenMP..... | 10 |
| 1.1.1 Первая программа – ряд Лейбница | 11 |
| 1.1.2 Основные конструкции параллельного выполнения ... | 17 |
| 1.1.3 Некоторые вспомогательные директивы | 22 |
| 1.1.4 Разделение данных | 24 |
| 1.1.5 Runtime-функции..... | 26 |
| 1.1.6 Вычисление определенного интеграла..... | 30 |
| 1.2 Интерфейс передачи сообщений MPI..... | 32 |
| 1.2.1 Краткое описание предоставляемых функций | 33 |
| 1.2.2 Снова ряд Лейбница | 36 |
| 1.2.3 Неравномерное распределение вычислений..... | 38 |
| 1.2.4 Умножение матрицы на вектор..... | 45 |
| 1.2.5 Перемножение матриц | 51 |
| Глава 2. Ярусно-параллельная форма программы | 60 |
| 2.1 Цель и механизм построения ЯПФ | 60 |
| 2.2 Реализация выполнения комплекса работ при использовании фиксированного количества параллельных ресурсов | 65 |
| 2.3 Случай неравномерной длительности работ | 69 |
| Глава 3. Сети конечных автоматов..... | 75 |
| 3.1 Программирование конечных автоматов | 75 |
| 3.2 Параллелизм сетей конечных автоматов | 80 |
| 3.3 Пример программной реализации | 83 |
| 3.3.1 Реализация с использованием OpenMP | 86 |

38. OpenMP C and C++ Application Program Interface (Version 1.0: October, 1998) [Электронный ресурс] URL: <http://www.openmp.org/mp-documents/cspec10.pdf>.
39. OpenMP C and C++ Application Program Interface (Version 2.0: March, 2002) [Электронный ресурс] URL: <http://www.openmp.org/mp-documents/cspec20.pdf>.
40. Wagner F., Schmuki R., Wagner T., Wolstenholme P. Modeling Software with Finite State Machines: A Practical Approach. – Auerbach Publications, 2006. – 392 p.
41. Wagner F. Moore or Mealy model? April 2005. [Электронный ресурс] URL: <http://www.stateworks.com/technology/TN10-Moore-Or-Mealy-Model/>.

рые свойства, присущие сетям конечных автоматов. В каком-то роде сеть Петри близка по смыслу к диаграмме состояний автомата, который может находиться одновременно в нескольких состояниях. Каждый переход обуславливается каким-либо подмножеством текущих состояний и заменяет его другим подмножеством состояний. Позиции и переходы сети Петри в такой интерпретации играют роль состояний автомата и действий на переходах соответственно.

Графически сеть Петри представляется в виде двудольного ориентированного графа с двумя типами вершин – позициями и переходами (рис. 23). Позиция обозначается кругом, переход – чертой или прямоугольником. Как правило, чертой обозначают простой, мгновенный переход, прямоугольником – длительный. В позициях могут находиться фишки – некая сущность, наличие которой говорит о том, что условие, соответствующее текущей позиции, выполняется. К примеру, наличие фишки может говорить о наличии входных данных для некоторой процедуры. Наличие нескольких фишек говорит о том, что условие выполнено с многократным запасом. В примере с процедурой это может быть наличие нескольких элементов в очереди ее входных данных. Количество фишек в каждой позиции является целым неотрицательным числом. Наличие фишек в позиции на графе обозначается числом либо жирными точками в соответствующем количестве. Совокупность всех фишек, размещенных в позициях сети Петри, называется разметкой сети.

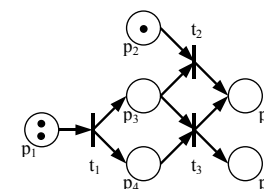


Рис. 23

Дуги сети Петри могут быть направлены лишь от позиций к переходам либо от переходов к позициям. Дуги могут быть кратными, т.е. иметь вес – также целое неотрицательное число. Наличие кратной дуги между позицией и переходом эквивалентно на-

личию между ними соответствующего количества простых дуг в том же направлении. Дуги, направленные от позиций к переходам, называются входными, направленные от переходов к позициям – выходными. Аналогичными терминами обозначаются и соответствующие позиции по отношению к некоторому переходу.

Таким образом, сеть Петри полностью описывается следующими параметрами:

- множество позиций;
- множество переходов;
- множество входных дуг (дуг от позиций к переходам);
- множество выходных дуг (дуг от переходов к позициям);
- множество фишек, размещенных в позициях изначально (начальная разметка).

Функционирование сети Петри осуществляется в виде последовательности срабатывания переходов. В каждый момент времени в сети есть некоторое количество разрешенных переходов, т.е. таких, которые могут сработать. Переход считается разрешенным, если во всех его входных позициях количество фишек не меньше, чем кратность соответствующих входных дуг. Из всего множества разрешенных переходов сработать может любой. Какой именно сработает, определяется вне сети, также как и момент его срабатывания во времени. Выбор может быть осуществлен на основе ожидания аппаратного события, события завершения длительного процесса, выбора пользователем и т.п. При срабатывании простого перехода из каждой его входной позиции изымается количество фишек, равное кратности соответствующей входной дуги, после чего к каждой выходной позиции добавляются фишки в соответствии с кратностью выходных дуг. После каждого срабатывания перехода множество разрешенных переходов в общем случае меняется, поскольку меняется текущая разметка сети. Считается, что сеть Петри «жива», если в ней есть разрешенные переходы. Если после срабатывания очередного перехода разрешенных переходов в сети не остается, она завершает выполнение.

В случае, когда позиция является входной для двух или бо-

при программной реализации алгоритмов логического управления // Автоматика и телемеханика. – 1996. №6. С.148-158; №7. С.144-169.

27. Шалыто А.А., Туккель Н.И. От тьюрингова программирования к автоматному // Мир ПК. – 2002, №2. – С.144-149.
28. Шалыто А.А., Туккель Н.И. Преобразование итеративных алгоритмов в автоматные // Программирование. – 2002. № 5. – С. 12-26.
29. Шалыто А.А., Туккель Н.И. Программирование с явным выделением состояний // Мир ПК. – 2001, №8, №9. – С.116-121; С.132-138.
30. Шалыто А.А., Туккель Н.И., Шамгунов Н.Н. Реализация рекурсивных алгоритмов на основе автоматного подхода // Телекоммуникации и информатизация образования. – 2002. № 5. – С. 72-99.
31. Шпаковский Г.И., Серикова Н.В. Программирование для многопроцессорных систем в стандарте MPI: Пособие. – Мн.: БГУ, 2002. – 323 с.
32. Элементы параллельного программирования / Вальковский В.А., Котов В.Е., Марчук А.Г., Миренков Н.Н. – М.: Радио и связь, 1983. – 240 с.
33. Lee Edward A. The problem with threads // IEEE Computer. – Vol. 39, № 5, May 2006. – P. 33-42.
34. Snir M., Otto S., Huss-Lederman S., etc. MPI – The Complete Reference: The MPI Core. – 2-nd edn. – Cambridge: MIT Press, 1998. – 426 p.
35. Snir M., Otto S., Huss-Lederman S., etc. MPI: The complete Reference. – MIT Press, Cambridge, Massachusetts, 1997. [Электронный ресурс] URL: <http://www.netlib.org/utk/papers/mpi-book/mpi-book.html>.
36. MPI: A Message-Passing Interface Standard. (Version 1.1: June, 1995) [Электронный ресурс] URL: <http://www.mcs.anl.gov/mpi/standard.html>.
37. MPI-2: Extensions to the Message-Passing Interface. [Электронный ресурс] URL: <http://www.mcs.anl.gov/mpi/standard.html>.

16. Ли Эдвард А. Проблемы с потоками // IEEE Computer, 2006. – Перевод с англ.: Петров А.В., 2007 [Электронный ресурс] URL: <http://www.softcraft.ru/parallel/pwt/index.shtml>.
17. Любченко В.С. К проблеме создания модели параллельных вычислений // Труды Третьей международной конференции «Параллельные вычисления и задачи управления» (РАСО'2006). Москва, 2-4 октября 2006 г. Институт проблем управления им. В.А. Трапезникова РАН. – М.: Институт проблем управления им. В.А. Трапезникова РАН, 2006. – С. 1359-1374.
18. Любченко В.С. К решению проблемы обедающих философов Дейкстры // Высокопроизводительные параллельные вычисления на кластерных системах: Материалы четвертого международного научно-практического семинара. – Самара: СГАУ, 2005. – С. 186-193.
19. Любченко В.С. Конечно-автоматная технология программирования // Труды международной научно-методической конференции «Телематика'2001». СПб.: СПбГИТМО(ТУ), 2001. – С. 127-128.
20. Немнюгин С.А., Стесик О.Л. Параллельное программирование для многопроцессорных вычислительных систем. Серия "Мастер программ". – СПб.: БХВ-Петербург, 2002. – 400 с.
21. Питерсон Дж. Теория сетей Петри и моделирование систем: Пер. с англ. – М.: Мир, 1984. – 264 с.
22. Трахтенброт Б. А., Барздин Я. М. Конечные автоматы (поведение и синтез). – М.: Наука, 1970. – 400 с.
23. Хоар Ч. Взаимодействующие последовательные процессы: Пер. с англ. – М.: Мир, 1989. – 264 с.
24. Шалыто А.А. SWITCH-технология. Алгоритмизация и программирование задач логического управления. – СПб.: Наука, 1998. – 628 с.
25. Шалыто А.А. Автоматное проектирование программ. Алгоритмизация и программирование задач логического управления // Известия Академии наук. Теория и системы управления. – №6. Ноябрь-Декабрь 2000. – С. 63-81.
26. Шалыто А.А. Использование граф-схем и графов переходов

лее разрешенных переходов, срабатывание любого из них и соответствующее уменьшение количества фишек в ней может запретить другие переходы. Такой ситуацией моделируются конфликты, когда для выполнения нескольких операций требуется использование общего ресурса. Пример такой ситуации виден на рис. 23. В самом начале работы сети разрешенным является один переход – t_1 . После его первого срабатывания разрешенными являются все три перехода сети. Если же теперь сработает, к примеру, переход t_2 , он запретит переход t_3 , и наоборот, срабатывание t_3 запретит переход t_2 , поскольку переходы t_2 и t_3 имеют общую входную позицию p_3 .

Срабатывание простого перехода считается мгновенным. Поскольку вероятность одновременного происхождения двух мгновенных событий равна нулю, два простых перехода не могут сработать одновременно [21]. Отсюда вытекает довольно парадоксальное свойство сетей Петри: притом, что они моделируют асинхронное выполнение параллельных процессов, сама по себе работа сети Петри происходит строго последовательно. Именно невозможностью одновременного срабатывания каких-либо переходов определяется асинхронная природа сетей Петри. Параллелизм же выполнения выражается в том, что в один момент времени разные участки сети могут отражать выполнение разных независимых процессов.

Мы не будем касаться вопросов анализа и верификации сетей Петри [11, 21], считая, что этот этап уже пройден. Помимо приведенного классического описания сетей Петри, существуют различные расширенные модели. К примеру, цветные сети Петри дополняются введением типов фишек и помогают избежать многократного дублирования фрагментов сети в сложных системах. Сети Петри с приоритетами добавляют к разрешенным переходам приоритеты и тем самым позволяют снизить недетерминированность срабатываний, ограничивая множество разрешенных переходов группой переходов с наивысшим приоритетом. Существуют и другие расширения, из которых мы рассмотрим строго иерархические сети [11].

4.1.2 Строго иерархические сети

Ранее уже звучало, что срабатывание простого перехода сети Петри мгновенно. Однако зачастую переходами моделируется выполнение каких-либо далеко не мгновенных операций. В этом случае выполнению операции может быть сопоставлен длительный переход. Для наглядности длительные переходы, в отличие от простых, часто обозначаются прямоугольником (рис. 24, слева).

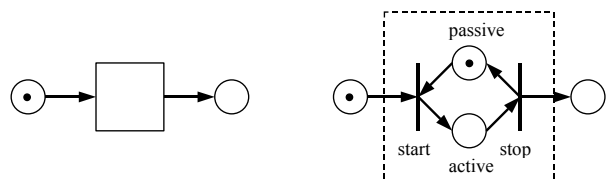


Рис. 24

Длительный переход, как и простой, начинает выполнение с изъятия фишек из входных позиций и завершает, соответственно, помещая фишки в выходные. Основное отличие от простого перехода заключается в том, что между этими двумя моментами могут срабатывать другие переходы. Иначе говоря, срабатывание длительного перехода не атомарно.

С момента начала выполнения до момента завершения длительный переход считается активным, в противном случае – пассивным. Длительный переход не может сработать, если он активен, т.е. уже выполняется, даже если он разрешен по наличию фишек во входных позициях.

Выполнение длительного перехода эквивалентно выполнению фрагмента сети, изображенного на рис. 24, справа. Здесь простые переходы start и stop характеризуют, соответственно, начало и завершение выполнения длительного перехода, позиция active характеризует активность длительного перехода, позиция passive – его пассивность.

Поскольку длительный переход не мгновенен и его срабатывание не атомарно, выполнение различных длительных переходов может перекрываться во времени, т.е. осуществляться параллельно.

Библиографический список

1. Антонов А.С. Введение в параллельные вычисления. Методическое пособие. – М.: Изд-во МГУ, 2002. – 70 с.
2. Антонов А.С. Параллельное программирование с использованием технологии MPI. Учебное пособие. – М.: Изд-во МГУ, 2004. – 71 с.
3. Букатов А.А., Дацюк В.Н., Жегуло А.И. Программирование многопроцессорных вычислительных систем. – Ростов-на-Дону: Изд-во ООО «ЦВВР», 2003. – 208 с.
4. Вавилов К.В. Программирование за... 1 (одну) минуту // Компьютер Price. – 2002. – № 31. – С. 288–293.
5. Вентцель Е.С. Исследование операций. – М.: Сов. радио, 1972. – 552 с.
6. Воеводин В.В., Воеводин Вл.В. Параллельные вычисления. Серия «Научное издание». – СПб.: БХВ-Петербург, 2002. – 608 с.
7. Глебов А.Н. Параллельное программирование в функциональном стиле. – 2003 [Электронный ресурс] URL: <http://www.softcraft.ru/parallel/ppfs.shtml>.
8. Глушков В.М. Синтез цифровых автоматов. – М.: Физматгиз, 1962. – 476 с.
9. Дехтярь М.И. Введение в схемы, автоматы и алгоритмы. – 2007 [Электронный ресурс] URL: <http://www.intuit.ru/departement/ds/introsaa/>.
10. Котов В.Е. Введение в теорию схем программ. – Новосибирск: Наука, 1978. – 258 с.
11. Котов В.Е. Сети Петри. – М.: Наука, 1984. – 160 с.
12. Кремер Н.Ш., Путко Б.А., Тришин И.М., Фридман М.Н. Исследование операций в экономике: Учебное пособие. Под. ред. Кремера Н.Ш. – М.: ЮНИТИ, 1997. – 408с.
13. Кузнецов Б.П. Психология автоматного программирования // ВУТЕ-Россия. – 2000. №11. – С. 22-29.
14. Кузнецов С.Д. Блеск и нищета легковесных процессов // Computerworld Россия. – 1996. №31.
15. Лескин А.А., Мальцев П.А., Спиридонов А.М. Сети Петри в моделировании и управлении. – Л.: Наука, 1989. – 133 с.

```

#pragma omp parallel for
for (int j = 0; j < m_mtxout[local].size(); j++)
    m_marking[j] += m_mtxout[local][j];
};

// сформируем набор локальных переходов,
// разрешение которых могло измениться
std::set<int> trchg;
// прежде всего, добавим сработавший локальный переход
trchg.insert(local);
// переберем все позиции, разметка которых изменилась
for (int j = 0; j < m_marking.size(); j++)
    if (m_marking[j] != marking_pre[j])
        // соберем все переходы, для которых они являются входными
        for (int i = 0; i < m_trlist.size(); i++)
            if (m_mtxin[i][j] > 0)
                trchg.insert(i);
// обновим с учетом сформированного набора
refresh_enabled(trchg);
}
};

// сеть Петри верхнего уровня
class petrinet_type: public transition_composite_type
{
public:

    typedef transition_composite_type::content_type content_type;

    // абстрактный тип среды, в которой "живет" сеть Петри
    class environment_abstract_type
    {
    public:
        // ожидание срабатывания одного из переданных переходов
        virtual int wait(const enabledlist_type &enabled) = 0;
    };

public:

    petrinet_type(const content_type &content):
        transition_composite_type(content)
    {}

    void live(environment_abstract_type &env)
    {
        activate();
        while (is_active())
            fire(env.wait(get_enabled()));
    }
};

```

Операция, выполняемая в течение длительного перехода, может содержать полный жизненный цикл какой-либо вложенной сети (рис. 25). На основе такого включения могут быть организованы иерархические сети произвольной вложенности. Если сеть не содержит дуг, соединяющих позиции и переходы разных уровней вложенности либо разных сетей одного уровня, то такая сеть называется строго иерархической [11]. В дальнейшем мы будем говорить лишь о строго иерархических сетях. Длительный переход, содержащий вложенную сеть, будем называть составным.

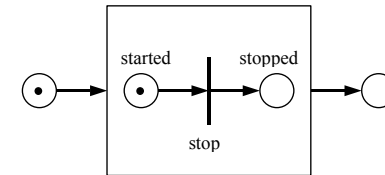


Рис. 25

В момент активации составного перехода из его входных позиций извлекаются фишки, после чего вложенная сеть начинает функционирование исходя из своей начальной разметки. Когда после очередного срабатывания внутреннего перехода вложенной сети в ней не остается разрешенных переходов, она прекращает работу. В этот момент соответствующий составной переход помещает фишки в выходные позиции и переходит в пассивное состояние. При следующей активации составного перехода вложенная сеть снова размечается в соответствии со своей начальной разметкой и начинает новый жизненный цикл.

Вследствие отсутствия связей внутренних позиций и переходов вложенной сети с внешними по отношению к ней, внутренняя структура вложенной сети скрыта от внешней и может быть реализована полностью независимо. С точки зрения внешней сети любой ее составной переход произвольной сложности, так же как и вообще любой длительный переход, могут быть представлены в виде составного перехода с простой вложенной сетью, изображенного на рис. 25. Здесь единственный простой переход характеризует завершение работы длительного перехода.

Именно этот факт позволяет нам довольно просто использовать сети Петри в параллельном программировании – длительные операции любого характера могут быть выполнены в виде составных переходов, которые могут выполняться параллельно.

4.1.3 Параллельные вычисления и синхронизация

Сетями Петри легко моделируется создание и выполнение параллельных ветвей различных вычислительных процессов. К примеру, на рис. 26 изображена одна из довольно популярных на сегодняшний день конструкций fork/join. Переход fork моделирует «разветвление» – создание из одной ветви выполнения двух параллельных ветвей. Это, как правило, реализуется путем создания одной дополнительной ветви вдобавок к существующей. Переход join, в свою очередь, осуществляет «слияние» двух ветвей по завершению их работы – уничтожение созданной параллельной ветви за ненадобностью.

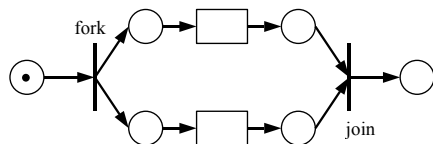


Рис. 26

Используемые сегодня объекты синхронизации также легко моделируются сетями Петри. К примеру, на рис. 27 изображен фрагмент сети Петри, моделирующий барьерную синхронизацию трех процессов.

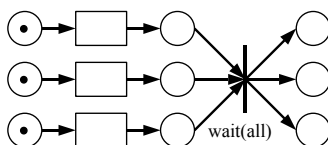


Рис. 27

В начале работы сети разрешенными являются три длительных перехода, характеризующих выполнение некоторых подзадач. Три независимых процесса выполняют эти подзадачи, и лишь после их полного завершения становится разрешенным пе-

```

m_pllist(content.get_pllist()),
m_trlist(content.get_trlist()),
m_mtxin(content.get_matrixin()),
m_mtxout(content.get_matrixout()),
m_marking_init(content.get_marking()),
m_enabled_or_active(m_trlist.size()),
m_offset(m_trlist.size())
{}

void activate(void)
{
    m_marking = m_marking_init;
    refresh_enabled();
}

bool is_active(void) const
{
    return (m_enabled_full.size() > 0);
}

enabledlist_type get_enabled(void) const
{
    return m_enabled_full;
}

void fire(int number)
{
    assert(number >= 0 && number < m_enabled_full.size());

    int local = m_location[number];
    int lower = number - m_offset[local];

    // запомним текущую разметку
    std::vector<int> marking_pre = m_marking;

    // если переход активен, выполним внутренний
    if (m_trlist[local]->is_active())
        m_trlist[local]->fire(lower);
    else
    {
        // иначе изыдем входные фишки
        #pragma omp parallel for
        for (int j = 0; j < m_mtxin[local].size(); j++)
            m_marking[j] -= m_mtxin[local][j];
        // активируем переход
        m_trlist[local]->on_activate();
        m_trlist[local]->activate();
    };
    // если переход перестал быть активным
    if (!m_trlist[local]->is_active())
    {
        m_trlist[local]->on_passivate();
        // выложим выходные фишки

```



```

refresh_enabled(trchg);
}

// частичное обновление списка разрешенных переходов
void refresh_enabled(const std::set<int> &trchg)
{
    // перебираем все переданные переходы
    std::set<int>::const_iterator it;
    for (it = trchg.begin(); it != trchg.end(); it++)
    {
        int i = *it;
        // если переход не активен, рассматриваем вопрос разрешения
        bool enabled = true;
        if (!m_trlist[i]->is_active())
        {
            #pragma omp parallel for reduction(&&: enabled)
            for (int j = 0; j < m_mtxin[i].size(); j++)
                enabled = enabled && (m_marking[j] - m_mtxin[i][j] >= 0);
        };
        m_enabled_or_active[i] = enabled;
    };

    // соберем все разрешенные переходы и их размещение
    m_enabled_full.clear();
    m_location.clear();
    for (int i = 0; i < m_trlist.size(); i++)
    {
        m_offset[i] = m_enabled_full.size();
        if (m_enabled_or_active[i])
        {
            // если переход разрешен и не активен, добавим его
            if (!m_trlist[i]->is_active())
            {
                m_enabled_full.push_back(m_trlist[i]);
                m_location.push_back(i);
            }
            else
            {
                // если активен, получаем внутренние разрешенные
                enabledlist_type in = m_trlist[i]->get_enabled();
                // добавляем в конец текущего списка
                m_enabled_full.insert(
                    m_enabled_full.end(), in.begin(), in.end());
                // формируем информацию о размещении
                m_location.insert(m_location.end(), in.size(), i);
            };
        };
    };
}

public:
transition_composite_type(const content_type &content):

```

реход-барьер wait(all). После его срабатывания все три процесса снова продолжают свою работу.

Похожим образом может быть смоделировано ожидание завершения любой из подзадач, первой завершившей свое выполнение. К примеру, на рис. 28 показан такой фрагмент сети. При завершении выполнения любого длительного перехода становится разрешенным переход wait(any). После его срабатывания дальнейшая работа продолжается, в то время как остальные подзадачи завершают свое выполнение. Дополнительная входная позиция перехода wait(any) защищает его от срабатывания при завершении остальных подзадач в случае, если в текущий момент ожидание не выполняется. В процессе дальнейшей работы сети фишка в эту позицию возвращается, и тем самым разрешается ожидание и обработка результатов выполнения остальных подзадач.

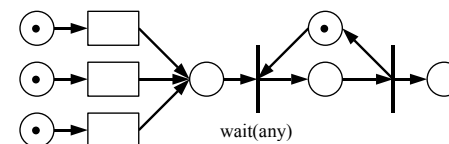


Рис. 28

Другой пример синхронизации параллельных процессов – критическая секция. Фрагмент сети на рис. 29 обеспечивает взаимноисключающий доступ к некоторому общему ресурсу для двух процессов.

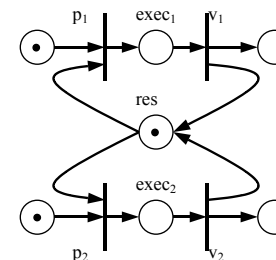


Рис. 29

Первый же процесс, захвативший доступ к критической секции, лишит фишки соответствующую позицию res, тем са-

мым запретив доступ к критической секции второму процессу до момента ее освобождения. В случае, когда в позиции *res* изначально более одной фишки, такой фрагмент сети моделирует использование семафора – объекта синхронизации, позволяющего одновременный доступ к ресурсу нескольких процессов в количестве не более заданного.

Важная особенность сетей Петри заключается в атомарности срабатывания перехода и, как следствие, возможности атомарного захвата одновременно нескольких ресурсов, что исключает возможность взаимоблокировки процессов (dead lock).

Рассмотрим такую ситуацию на примере. Допустим, имеется два процесса, каждому из которых необходим одновременный доступ к двум общим ресурсам. Попробуем реализовать это с помощью критических секций. Вариант такой реализации, описанный в виде фрагмента сети Петри, изображен на рис. 30.

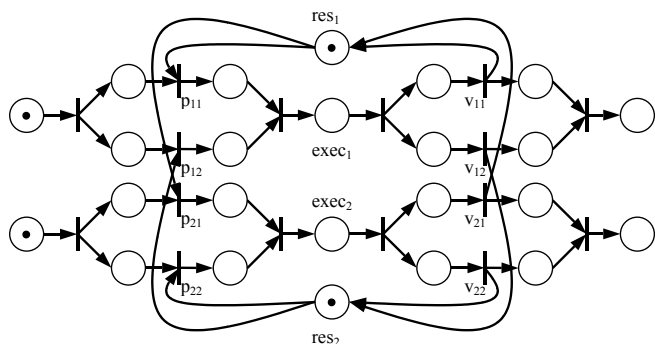


Рис. 30

Для выполнения некоторой операции каждому из двух процессов требуются два ресурса, доступность которых характеризуют фишки в позициях *res*₁ и *res*₂. Каждый процесс пытается захватить оба ресурса в произвольной последовательности. Если переходы *p*₁₁ и *p*₁₂ в произвольной последовательности сработают раньше, чем переходы *p*₂₁ и *p*₂₂, либо наоборот, оба процесса бла-

```

}

// получение начальной разметки
marking_type get_marking(void) const
{
    marking_type marking(m_plmap.size());
    for (int i = 0; i < marking.size(); i++)
    {
        tokmap_type::const_iterator it = m_tokmap.find(i);
        marking[i] = (it != m_tokmap.end()) ? it->second : 0;
    };
    return marking;
}

private:

// список позиций
placelist_type m_pllist;

// список переходов
transitionlist_type m_trlist;

// матрицы входных и выходных дуг
arcmatrix_type m_mtxin;
arcmatrix_type m_mtxout;

// начальная разметка
marking_type m_marking_init;

// текущая разметка
marking_type m_marking;

// набор состояний локальных переходов - разрешены или активны
std::vector<bool> m_enabled_or_active;

// полный список разрешенных сейчас переходов, включая внутренние
enabledlist_type m_enabled_full;

// размещение этих переходов по локальным номерам
std::vector<int> m_location;

// смещение начала области каждого локального перехода
std::vector<int> m_offset;

// полное обновление списка разрешенных переходов
void refresh_enabled(void)
{
    // сформируем набор всех переходов
    std::set<int> trchg;
    for (int i = 0; i < m_trlist.size(); i++)
        trchg.insert(i);
    // передадим на частичное обновление

```

```

    add_arc(in, tr, weight, m_inmap);
}
// добавление выходной дуги
void add_arc(
    transition_abstract_type &tr,
    place_type &out,
    int weight = 1)
{
    add_arc(out, tr, weight, m_outmap);
}

// добавление некоторого количества фишек к позиции
void add_token(place_type &pl, int tokens = 1)
{
    assert(m_plmap.find(&pl) != m_plmap.end());
    assert(tokens > 0);

    if (m_tokmap.find(m_plmap[&pl]) == m_tokmap.end())
        m_tokmap.insert(tokmap_type::value_type(m_plmap[&pl], tokens));
    else
        m_tokmap[m_plmap[&pl]] += tokens;
}

// ----- функции, используемые сетью -----

// получение списка позиций
placelist_type get_pllist(void) const
{
    placelist_type pllist(m_plmap.size());
    plmap_type::const_iterator it;
    for (it = m_plmap.begin(); it != m_plmap.end(); it++)
        pllist[it->second] = it->first;
    return pllist;
}

// получение списка переходов
transitionlist_type get_trlist(void) const
{
    transitionlist_type trlist(m_trmap.size());
    trmap_type::const_iterator it;
    for (it = m_trmap.begin(); it != m_trmap.end(); it++)
        trlist[it->second] = it->first;
    return trlist;
}

// получение матриц входных и выходных дуг
arcmatrix_type get_matrixin(void) const
{
    return build_matrix(m_inmap);
}
arcmatrix_type get_matrixout(void) const
{
    return build_matrix(m_outmap);
}

```

гополучно закончат свое выполнение. Если же первыми сработают переходы p_{11} и p_{22} , либо p_{12} и p_{21} , каждый процесс окажется навсегда заблокированным в ожидании другого.

Разумеется, в данном случае проблема может быть решена путем ухода от произвольной последовательности захвата ресурсов и введения жесткого порядка. К примеру, можно обязать оба процесса пытаться сначала захватить ресурс, связанный с позицией res_1 . Однако такой подход существенно усложняется при усложнении задачи в целом. В то же время на рис. 31 мы видим фрагмент сети Петри, успешно решающий поставленную задачу без введения порядка захвата – захват ресурсов осуществляется одновременно в момент срабатывания переходов pp_1 или pp_2 .

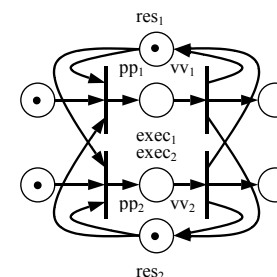


Рис. 31

В последнем примере видны преимущества сетей Петри перед популярными сегодня в программировании средствами синхронизации. С помощью сетей Петри оказывается гораздо проще организовать сложное взаимодействие различных участков системы, особенно в ситуациях, когда в разрабатываемой системе происходит широкое использование разделяемых ресурсов.

4.1.4 Задача об обедающих философах

Одной из классических задач, иллюстрирующих проблемы синхронизации процессов, является задача об обедающих философах. Ее постановка довольно проста и вкратце заключается в следующем. За круглым столом сидят пятеро философов, перед ними стоит блюдо спагетти. На столе лежат пять вилок – по одной между каждыми двумя соседними философами. По условию каждому философу для еды необходимо две вилки – лежащие не-

посредственно слева и справа от него. Каждый философ пребывает за столом в одном из двух состояний – размышляет или ест. В последнем случае оба его ближайших соседа размышляют, поскольку для еды им не хватает вилок.

Основная проблема, иллюстрируемая этой задачей – проблема возможности взаимоблокировки, которая была описана раньше. В случае реализации с последовательным захватом вилок возможна ситуация, когда одновременно все философы возьмут, к примеру, левую от себя вилку. Тогда ни один из них не сможет взять правую вилку, и каждый окажется заблокированным в ожидании ее освобождения.

Как уже говорилось выше, проблема взаимоблокировки легко решается сетями Петри, поскольку они предоставляют возможность атомарного захвата одновременно нескольких ресурсов.

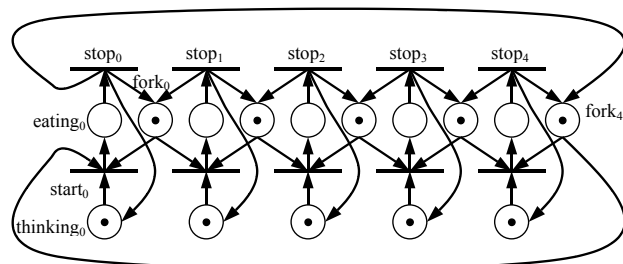


Рис. 32

На рис. 32 изображена сеть Петри, решающая задачу об обедающих философях [21]. Позиции eating и thinking здесь характеризуют пребывание соответствующего философа в состоянии еды или размышлений. Позициями fork обозначается доступность вилок. Переходы start и stop характеризуют переход философа к еде и к размышлениям соответственно.

Однократный процесс приема пищи каждым философом представляет собой длительную операцию и может быть представлен в виде длительного составного перехода, изображенного на рис. 25. В этом случае срабатывание перехода stop характеризует завершение приема пищи философом и, соответственно, завершение длительного перехода. На рис. 33 изо-

```

assert(m_plmap.find(&pl) != m_plmap.end());
assert(m_trmap.find(&tr) != m_trmap.end());
assert(weight > 0);

arcmap_type::key_type arc(m_trmap[&tr], m_plmap[&pl]);
if (arcmap.find(arc) == arcmap.end())
    arcmap.insert(arcmap_type::value_type(arc, weight));
else
    arcmap[arc] += weight;
}

// построение матрицы кратности дуг
arcmatrix_type build_matrix(const arcmap_type &arcmap) const
{
    arcmatrix_type matrix(m_trmap.size());
    for (int i = 0; i < matrix.size(); i++)
    {
        std::vector<int> vec(m_plmap.size());
        for (int j = 0; j < vec.size(); j++)
        {
            arcmap_type::const_iterator it;
            it = arcmap.find(arcmap_type::key_type(i, j));
            vec[j] = (it != arcmap.end()) ? it->second : 0;
        };
        matrix[i] = vec;
    };
    return matrix;
}

public:

// ----- функции, используемые при построении сети -----

// добавление позиции
void add_place(place_type &pl)
{
    assert(m_plmap.find(&pl) == m_plmap.end());
    m_plmap.insert(plmap_type::value_type(&pl, m_plmap.size()));
}

// добавление перехода
void add_transition(transition_abstract_type &tr)
{
    assert(m_trmap.find(&tr) == m_trmap.end());
    m_trmap.insert(trmap_type::value_type(&tr, m_trmap.size()));
}

// добавление входной дуги
void add_arc(
    place_type &in,
    transition_abstract_type &tr,
    int weight = 1)
{

```

```

enabledlist_type get_enabled(void) const
{ return enabledlist_type(); }

void fire(int number) { assert(false); }

};

// составной переход (сеть Петри)
class transition_composite_type: public transition_abstract_type
{
private:

    typedef std::vector<place_type *> placelist_type;
    typedef std::vector<transition_abstract_type *>
        transitionlist_type;
    typedef std::vector<int> marking_type;
    typedef std::vector<marking_type> arcmatrix_type;

public:

    // содержимое сети Петри
    class content_type
    {
    private:

        typedef std::map<place_type *, int> plmap_type;
        typedef std::map<transition_abstract_type *, int> trmap_type;
        typedef std::map<std::pair<int, int>, int> arcmap_type;
        typedef std::map<int, int> tokmap_type;

        // отображение позиций на их номера
        plmap_type m_plmap;

        // отображение переходов на их номера
        trmap_type m_trmap;

        // отображения входных и выходных дуг на их веса
        arcmap_type m_inmap, m_outmap;

        // отображение номеров позиций на количество фишек
        tokmap_type m_tokmap;

    private:

        // добавление дуги некоторой кратности
        void add_arc(
            place_type &pl,
            transition_abstract_type &tr,
            int weight,
            arcmap_type &arcmap)
        {

```

бражена сеть с использованием таких составных переходов, содержимое которых здесь опущено для компактности.

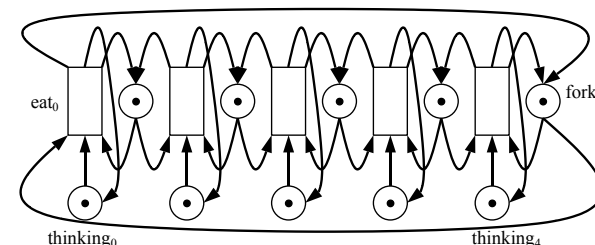


Рис. 33

Помимо проблемы взаимоблокировки, задача об обедающих философах демонстрирует возможность голодания (заговора соседей), т.е. такую ситуацию, в которой один или более философов никогда не получают доступ к еде. Проблема голодания легко решается не только сетями Петри, поэтому их применение здесь не столь иллюстративно, как решение проблемы взаимоблокировки, однако вскользь мы рассмотрим и ее.

Одним из довольно простых решений этой проблемы, хотя, конечно, далеко не самым лучшим, является следующее. К сети, изображенной на рис. 33, добавим барьерную синхронизацию после того, как каждый философ осуществит прием пищи по одному разу. Для этого добавим к каждому философу по две позиции – позицию `todo`, количество фишек в которой отражает количество предстоящих подходов философа к еде до следующего барьера, и позицию `done`, отражающую количество выполненных подходов. Полученная сеть изображена на рис. 34.

В случае если синхронизация после каждого подхода к еде является слишком частой мерой, количество фишек в каждой позиции `todo` может быть увеличено. Более того, начальные количества фишек в этих позициях могут быть заданы разными, в зависимости от потребности в интенсивном питании каждого из философов. В соответствии с начальным количеством фишек в позициях `todo` должна быть изменена и крат-

ность дуг, ведущих в переход-барьер и из него.

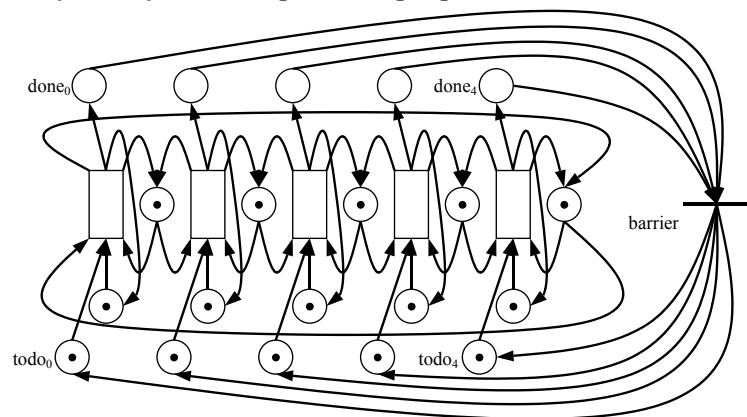


Рис. 34

С учетом такой модификации сети каждый философ будет получать доступ к еде в обычном порядке, пока не закончатся фишки в соответствующей позиции *todo*, после чего остановится на размышлениях. Таким образом, пока каждый из пяти философов не получит доступ к еде заданное количество раз, переход к следующему циклу осуществлен не будет. Когда все фишки из позиций *todo* постепенно переместятся в соответствующие позиции *done*, переход-барьер снова переместит их в позиции *todo*, после чего начнется следующий цикл приема пищи.

4.2 Пример реализации механизма сетей Петри

Одним из подходов, используемых при внедрении сетей Петри в программирование, является отождествление переходов с операторами программы, а позиций – с их готовностью к выполнению. Здесь возникает соблазн пренебречь длительностью простых операций и представить программу в виде одноуровневой сети, формально содержащей только простые переходы. Однако при разработке параллельной программы в этом случае необходимо учитывать тот факт, что вследствие того, что срабатывания простых переходов последовательны, т.е. не могут перекрывать друг друга по времени, организованная таким образом

Приложение 4. Классы построения и выполнения сетей Петри

Приводится набор классов, предоставляющий возможность построения и выполнения рассмотренных в четвертой главе сетей Петри и иерархических сетей. Там же рассматриваются вопросы распараллеливания программ, построенных на основе таких классов.

```
// пустой тип для обозначения позиции
class place_type {};

// абстрактный тип перехода
class transition_abstract_type
{
public:

    typedef std::vector<transition_abstract_type *> enabledlist_type;

    // активация перехода
    virtual void activate(void) = 0;

    // получение информации, активен ли переход
    virtual bool is_active(void) const = 0;

    // получение списка внутренних разрешенных переходов
    // (только если текущий переход активен)
    virtual enabledlist_type get_enabled(void) const = 0;

    // срабатывание одного из разрешенных внутренних переходов
    // (только если текущий переход активен)
    virtual void fire(int number) = 0;

    // обработчики событий
    virtual void on_activate(void) {}
    virtual void on_passivate(void) {}

};

// простой (атомарный) переход
class transition_simple_type: public transition_abstract_type
{
public:

    void activate(void) {}

    bool is_active(void) const { return false; }
```



```

    m_factory.destroy_fsm(i, m_pfsm[i]);
}

int number_input(void) const
{
    return m_factory.number_input();
}

int number_output(void) const
{
    return m_factory.number_output();
}

void put_input(const std::vector<data_type> &input)
{
    m_shared.put_input(input);
}

void do_work(void)
{
    // прочитать входные данные, выполнить действия
    #pragma omp parallel
    {
        #pragma omp for
        for (int i = 0; i < m_pfsm.size(); i++)
        {
            m_pfsm[i]->put_input(m_shared.get_input(i));
            m_pfsm[i]->do_work();
        };
        // записать выходные (запись должна быть отделена от чтения)
        #pragma omp for
        for (int i = 0; i < m_pfsm.size(); i++)
            m_shared.put_output(i, m_pfsm[i]->get_output());
    };
}

std::vector<data_type> get_output(void) const
{
    return m_shared.get_output();
}

bool is_off(void) const
{
    // сеть в начальном состоянии, когда все автоматы в начальном
    bool rc = true;
    #pragma omp parallel for reduction(&&: rc)
    for (int i = 0; i < m_pfsm.size(); i++)
        rc = (rc && m_pfsm[i]->is_off());
    return rc;
}
};

```

программа не может быть параллельной. Поэтому при реализации параллельных программ мы будем рассматривать иерархические сети с составными переходами.

4.2.1 Функционирование строго иерархических сетей

При реализации механизма работы сети Петри мы будем исходить из того факта, что все срабатывания и соответствующие изменения разметки последовательны. Таким образом, работа сетей, в том числе иерархических, будет нами реализована в виде последовательной программы. При этом, однако, на ее основе могут быть построены параллельные программы, процесс выполнения которых отражается выполнением соответствующих сетей.

В приложении 4 приведен используемый в качестве примера исходный код классов, реализующих работу иерархических сетей. Среди них объявляется класс позиции `place_type`, а также абстрактный класс перехода `transition_abstract_type`. На основе последнего строятся класс простого перехода `transition_simple_type` и класс составного перехода `transition_composite_type`. Абстрактным классом перехода описывается следующий предоставляемый каждым переходом интерфейс:

```

class transition_abstract_type
{
public:
    typedef std::vector<transition_abstract_type *> enabledlist_type;

    // активация перехода
    virtual void activate(void) = 0;
    // получение информации, активен ли переход
    virtual bool is_active(void) const = 0;
    // получение списка внутренних разрешенных переходов
    // (только если текущий переход активен)
    virtual enabledlist_type get_enabled(void) const = 0;
    // срабатывание одного из разрешенных внутренних переходов
    // (только если текущий переход активен)
    virtual void fire(int number) = 0;

    // обработчики событий
    virtual void on_activate(void) {}
    virtual void on_passivate(void) {}
};

```

Функция `activate` переводит переход в активное состояние. С

помощью функции `is_active` может быть осуществлена проверка, находится ли переход все еще в активном состоянии. Если переход активен, т.е. имеет внутренние разрешенные переходы, список этих переходов возвращается функцией `get_enabled`. При этом возвращается полный список всех переходов, включая внутренние переходы активных составных переходов всех уровней вложенности. Наконец, функция `fire` выполняет срабатывание конкретного перехода из списка, полученного с помощью функции `get_enabled`. На вход ей передается номер позиции перехода в этом списке. Функции `get_enabled` и `fire` должны вызываться лишь в случае, когда функция `is_active` возвращает значение истины.

Помимо функций управления переходом, интерфейс `transition_abstract_type` описывает два обработчика, вызываемых при переходе в активное и пассивное состояние. Эти обработчики при желании могут быть переопределены в унаследованных классах.

Класс простого перехода `transition_simple_type` предоставляет реализацию перечисленных функций с учетом специфики простого перехода. Простой переход активен лишь мгновение, т.е. при проверке состояния всегда пассивен. Помимо этого, простой переход не содержит вложенной сети, поэтому возвращает пустой список внутренних разрешенных переходов, функция же срабатывания внутреннего перехода бессмысленна.

Класс составного перехода `transition_composite_type` реализует работу одноуровневой сети Петри. Внутри него объявляется класс содержимого составного перехода `content_type`, который введен для удобства заполнения перехода внутренними элементами. Структуры, в которых удобно хранить информацию о внутренних элементах при заполнении, отличаются от тех, которые удобно использовать при выполнении составного перехода, вследствие чего в какой-то момент необходимо преобразование одних структур в другие. Именно с этой целью введен класс `content_type`, чтение и преобразование структур которого производится в момент создания объекта составного перехода.

Класс `content_type` хранит указатели на позиции и переходы

```
// сохранение входных данных сети
void put_input(const std::vector<data_type> &input)
{
    copy(input.begin(), input.end(),
          m_data.begin() + m_outpos.back());
}

// восстановление входных данных конкретного автомата
std::vector<data_type> get_input(int i) const
{
    std::vector<data_type> input;
    for (int j = 0; j < m_inpos[i].size(); j++)
        input.push_back(m_data[m_inpos[i][j]]);
    return input;
}

// восстановление выходных данных сети
std::vector<data_type> get_output(void) const
{
    std::vector<data_type> output;
    for (int j = 0; j < m_inpos.back().size(); j++)
        output.push_back(m_data[m_inpos.back()[j]]);
    return output;
}
};

private:

// список активных автоматов
std::vector<fsm_abstract_type *> m_pfsm;
// фабрика сети
factory_abstract_type &m_factory;
// общая область памяти
shared_area_type m_shared;

public:

fsmnet_type(factory_abstract_type &factory):
    m_factory(factory), m_shared(m_factory.get_links().get())
{
    // создание автоматов
    m_pfsm.resize(m_factory.number_fsm());
    #pragma omp parallel for
    for (int i = 0; i < m_pfsm.size(); i++)
        m_pfsm[i] = m_factory.create_fsm(i);
}

~fsmnet_type(void)
{
    // уничтожение автоматов
    #pragma omp parallel for
    for (int i = m_pfsm.size() - 1; i >= 0; i--)
```

```

    idxout : outsize.size() - 1;
    idxin = (idxin != PSEUDOFSM_NETOUTPUT) ?
    idxin : insize.size() - 1;

    outsize[idxout] =
    std::max(outsize[idxout], links[i].first.second);
    insize[idxin] =
    std::max(insize[idxin], links[i].second.second);
};
// корректируем до размеров
for (int i = 0; i < outsize.size(); i++)
    outsize[i]++;
for (int i = 0; i < insize.size(); i++)
    insize[i]++;

// формируем список позиций начал областей выходов
int fullsize = 0;
for (int i = 0; i < outsize.size(); i++)
{
    m_outpos.push_back(fullsize);
    fullsize += outsize[i];
};
// создаем массив общих данных
m_data.resize(fullsize);
fill(m_data.begin(), m_data.end(), 0);
// сохраняем размеры выходных областей в массиве общих данных
m_outsize.assign(outsize.begin(), outsize.end());

// создаем список списков позиций входов
for (int i = 0; i < insize.size(); i++)
    m_inpos.push_back(std::vector<int>(insize[i]));

// заполняем список списков позиций входов
for (int i = 0; i < links.size(); i++)
{
    int idxout = links[i].first.first;
    int idxin = links[i].second.first;
    idxout = (idxout != PSEUDOFSM_NETINPUT) ?
    idxout : m_outpos.size() - 1;
    idxin = (idxin != PSEUDOFSM_NETOUTPUT) ?
    idxin : m_inpos.size() - 1;
    int out = links[i].first.second;
    int in = links[i].second.second;

    m_inpos[idxin][in] = m_outpos[idxout] + out;
};
}

// сохранение выходных данных конкретного автомата
void put_output(int i, const std::vector<data_type> &output)
{
    copy(output.begin(), output.end(), m_data.begin() + m_outpos[i]);
}

```

сети вместе с их номерами в соответствии с порядком их добавления. Помимо этого, в нем хранится информация о входных и выходных дугах, а также о начальной разметке. Класс предоставляет вызывающему коду функции добавления позиции, перехода, входной и выходной дуг, а также помещения фишки в позицию. Помимо этого, для класса составного перехода предоставляются функции получения в упорядоченной форме списка позиций, списка переходов, матриц входных и выходных позиций и вектора начальной разметки. Все эти данные передаются в соответствующие структуры класса составного перехода в момент создания соответствующего объекта.

В процессе выполнения составной переход постоянно содержит текущий полный список внутренних разрешенных переходов `m_enabled_full`. Для поддержания актуальности этого списка используются две функции `refresh_enabled`, о которых подробнее поговорим позже. Помимо этого, в объекте содержится вектор текущей разметки `m_marking`, набор флагов разрешения либо активности переходов текущего уровня (без учета вложенности), а также две структуры данных, используемых для локализации срабатывающего перехода. Функция `activate` размечает переход в соответствии с начальной разметкой и обновляет текущее состояние списка разрешенных переходов. Если этот список не пуст, при вызове функции `is_active` возвращается значение истины. Построенный список может быть получен с помощью функции `get_enabled`. Наконец, функция `fire` локализует переход, номер которого передан ей в качестве параметра, и осуществляет действия по срабатыванию. Подробнее об этом мы поговорим после обсуждения функций обновления списка разрешенных переходов.

Определение, разрешен ли какой-либо переход, выполняется на основе матрицы входных дуг и текущей разметки. При большом количестве позиций эта операция может быть довольно тяжелой, поэтому следует избегать частого вычисления разрешения для всех переходов. По этой причине мы имеем две функции обновления текущего списка разрешенных переходов. Одна из них выполняет полное обновление списка разрешенных

переходов и вызывается из функции активации составного перехода activate, другая – частичное обновление, при котором анализируются лишь переходы, попадающие в переданное в качестве параметра множество. Последняя функция вызывается после каждого срабатывания внутреннего перехода из функции fire.

В задачу функции полного обновления входит построение множества номеров переходов, включающего все переходы текущего уровня (далее – локальные переходы), после чего она вызывает функцию частичного обновления с этим множеством в качестве параметра.

Функция частичного обновления первым делом вычисляет флаги разрешения или активности локальных переходов с переданными номерами. Результаты попадают в набор флагов m_enabled_or_active, в котором после предыдущих обновлений уже содержатся значения относительно локальных переходов, не попавших в переданное множество. Значения в этом списке флагов говорят о том, требует ли соответствующий переход внимания при построении полного списка разрешенных переходов m_enabled_full.

Как было сказано ранее, если переход активен, он не может сработать, даже если он разрешен. Поэтому, при построении полного списка нас интересует либо сам переход (если он разрешен и пассивен), либо его внутренние разрешенные переходы (если он активен). Переход может быть активен, только если он составной, а значит, содержит непустой список внутренних разрешенных переходов. Полный список строится путем последовательного соединения локальных разрешенных переходов и внутренних списков локальных активных переходов. В процессе построения полного списка осуществляется также сохранение в векторе m_location номеров локальных переходов, которым соответствуют конкретные элементы полного списка. В векторе m_offset сохраняются смещения областей каждого локального перехода от начала списка m_enabled_full. Обе эти структуры нужны для локализации сработавшего перехода по его позиции в полном списке.

В самом начале функции fire осуществляется локализация

```
virtual int number_input(void) const = 0;

// количество выходных каналов сети
virtual int number_output(void) const = 0;

// получение списка всех связей
virtual linkstore_type get_links(void) const = 0;

// создание i-го автомата в сети
virtual fsm_abstract_type *create_fsm(int i) = 0;

// уничтожение его же
virtual void destroy_fsm(int i, fsm_abstract_type *pfsm) = 0;

};

private:

// класс для работы с общими данными
class shared_area_type
{
    // общие данные (выходы всех автоматов и входы сети)
    std::vector<data_type> m_data;
    // позиции и размеры областей выходов автоматов и входов сети
    std::vector<int> m_outpos, m_outsize;
    // позиции входов автоматов и выходов сети
    std::vector<std::vector<int> > m_inpos;

public:
    // в конструкторе на основе входящего списка связей формируем
    // область общих данных, список позиций выходных областей
    // автоматов, а также списки позиций конкретных входов автоматов
    shared_area_type(const std::vector<link_type> &links)
    {
        // количество автоматов = максимальный указанный + 1
        int fsmnum = 0;
        for (int i = 0; i < links.size(); i++)
        {
            fsmnum = std::max(fsmnum, links[i].first.first);
            fsmnum = std::max(fsmnum, links[i].second.first);
        };
        fsmnum++;

        // кол-ва выходов автоматов (сначала) и входов сети (последний)
        std::vector<int> outsize(fsmnum + 1);
        // кол-ва входов автоматов (сначала) и выходов сети (последний)
        std::vector<int> insize(fsmnum + 1);
        // ищем максимальные номера входов и выходов для всех
        for (int i = 0; i < links.size(); i++)
        {
            int idxout = links[i].first.first;
            int idxin = links[i].second.first;
            idxout = (idxout != PSEUDOFSM_NETINPUT) ?
```

```

    if (it != m_handlertable.end())
        m_state = (this->*(it->second))(m_state);
}

std::vector<data_type> get_output(void) const
{
    return m_output;
}

bool is_off(void) const
{
    // проверка завершения работы
    return (m_state == STATE_OFF);
}
};

// сеть конечных автоматов
class fsmnet_type: public fsm_abstract_type
{
public:

    typedef std::pair<std::pair<int, int>, std::pair<int, int> >
        link_type;
    enum {PSEUDOFSM_NETINPUT = -1, PSEUDOFSM_NETOUTPUT = -2};

    // этот класс - для упрощения добавления связей
    class linkstore_type
    {
    private:
        std::vector<link_type> m_links;
    public:
        void add(int srcfsm, int srcnum, int dstfsm, int dstnum)
        {
            m_links.push_back(link_type(
                link_type::first_type(srcfsm, srcnum),
                link_type::second_type(dstfsm, dstnum)));
        }
        std::vector<link_type> get(void) const
        {
            return m_links;
        }
    };

    // абстрактный тип фабрики автоматной сети
    class factory_abstract_type
    {
    public:

        // количество автоматов в сети
        virtual int number_fsm(void) const = 0;

        // количество входных каналов сети

```

перехода, позиция которого передана в качестве параметра, а именно определение локального перехода, к области которого относится этот переход, и позиции в его полном списке внутренних разрешенных переходов. Последний параметр имеет смысл, если соответствующий локальный переход окажется активным. Если же он пассивен, значит, он просто разрешен и представлен в списке лишь одним элементом.

После определения локального перехода функция запоминает текущую разметку для анализа ее изменений в дальнейшем. Затем осуществляются основные действия по срабатыванию. Если переход активен, то вычисленная позиция в его внутреннем списке передается его функции срабатывания. Если же пассивен, осуществляются действия по изъятию фишек из его входных позиций и активации перехода. В этот момент пассивный переход может стать активным или остаться пассивным, если это простой переход. Активный переход после осуществления внутреннего срабатывания также может остаться активным или завершить работу и перейти в пассивное состояние. Поэтому следующим этапом выполняется проверка активности локального перехода. Если переход пассивен, осуществляются действия по деактивации и помещаются фишки в выходные позиции.

Наконец, в конце функции срабатывания осуществляется частичное обновление текущего списка разрешенных переходов. Для этого строится множество номеров подлежащих рассмотрению локальных переходов. В это множество попадает сработавший локальный переход, а также все переходы, разметка входных позиций которых изменилась в течение текущего срабатывания. Изменения определяются на основе предыдущей разметки, запомненной в начале функции.

При рассмотренном подходе приходится мириться с тем, что смена разметки сети при срабатывании перехода не является мгновенной операцией, вследствие чего само по себе программное выполнение большой сети может занимать значительное количество вычислительных ресурсов. Однако в случае строго иерархических сетей отдельные составные переходы с крупными вложенными сетями, как уже говорилось выше, могут быть пред-

ставлены в виде составного перехода, изображенного на рис. 25 и содержащего простую сеть, срабатывание единственного перехода которой говорит о завершении работы исходной вложенной сети. Такой подход в нашем случае позволяет возложить задачу выполнения чересчур сложных составных переходов в виде автономных сетей на другие параллельные ресурсы. Вопросы конкретной реализации параллельного выполнения будут рассмотрены позже.

Наконец, класс `petrinet_type` реализует работу всей иерархической сети. Поскольку вся сеть является частным случаем составного перехода, класс сети Петри наследуется от класса `transition_composite_type`. Класс определяет функцию выполнения жизненного цикла сети `live`, в которой осуществляется активация соответствующего составного перехода и последовательность срабатываний внутренних переходов до тех пор, пока переход не перестанет быть активен. Выбор срабатывающего перехода каждый раз осуществляется извне с помощью переданного объекта окружения, реализующего интерфейс `environment_abstract_type`. Этот интерфейс содержит одну функцию, которая принимает полный список разрешенных переходов сети, ожидает срабатывания любого из них и возвращает номер соответствующей позиции в переданном списке.

Классы, реализующие интерфейс `environment_abstract_type`, могут быть довольно разными, в зависимости от решаемой задачи. Одним из самых простых вариантов, зачастую используемых при моделировании, является ожидание срабатывания на основе случайного выбора. Каждому переходу может быть сопоставлена своя интенсивность срабатывания, на основе которой вычисляется вероятность срабатывания среди конкретного набора разрешенных переходов. Мы же приведем вариант реализации для случая одинаковой интенсивности срабатывания всех переходов:

```
class environment_random_type: public
    petrinet_type::environment_abstract_type
{
public:
    // инициализация датчика псевдослучайных чисел
    static void srandom(void)
    {
        srand((unsigned) time(NULL));
    }
};
```

```
// количество входов и выходов автомата
int m_inputnum, m_outputnum;

// состояние
state_type m_state;

// таблица обработчиков
typedef std::map<state_type, handler_type> handlertable_type;
handlertable_type m_handlertable;

protected:

// входные и выходные параметры автомата
std::vector<data_type> m_input, m_output;

protected:

void add_handler(state_type state, handler_type handler)
{
    m_handlertable.insert(
        handlertable_type::value_type(state, handler));
}

public:

fsm_type(int inputnum, int outputnum):
    m_state(STATE_OFF),
    m_inputnum(inputnum), m_outputnum(outputnum),
    m_input(m_inputnum), m_output(m_outputnum)
{
}

int number_input(void) const
{
    return m_inputnum;
}

int number_output(void) const
{
    return m_outputnum;
}

void put_input(const std::vector<data_type> &input)
{
    m_input.assign(input.begin(), input.end());
}

void do_work(void)
{
    // найти обработчик по текущему состоянию и вызвать его
    handlertable_type::iterator it = m_handlertable.find(m_state);
```


Приложение 3. Классы построения и выполнения сетей конечных автоматов

Приводится последовательная версия классов, используемых в третьей главе для построения и выполнения автоматных сетей, содержащая директивы распараллеливания OpenMP.

```
// абстрактный тип автомата
class fsm_abstract_type
{
public:

    // тип входных и выходных данных
    typedef int data_type;

    // тип состояния автомата
    typedef int state_type;

    // количество входных каналов
    virtual int number_input(void) const = 0;

    // количество выходных каналов
    virtual int number_output(void) const = 0;

    // передать автомату вектор входных данных
    virtual void put_input(const std::vector<data_type> &input) = 0;

    // выполнить такт работы автомата
    virtual void do_work(void) = 0;

    // получить от автомата вектор выходных данных
    virtual std::vector<data_type> get_output(void) const = 0;

    // проверка, находится ли автомат в начальном состоянии
    virtual bool is_off(void) const = 0;

};

// конечный автомат
class fsm_type: public fsm_abstract_type
{
protected:

    // обработчик, вызываемый из do_work в конкретном состоянии
    typedef state_type (fsm_type::*handler_type)(state_type state);
    // начальное/конечное состояние
    enum {STATE_OFF = -1};

private:
```

```
}
// генерация псевдослучайного числа в интервале [0, bound)
static int random(int bound)
{
    assert(bound > 0);
    return int(rand() / (RAND_MAX + 1.0) * bound);
}
// срабатывает произвольный переход
int wait(const petrinet_type::enabledlist_type &enabled)
{
    return random(enabled.size());
}
};
```

С использованием описанных классов создание и выполнение сети Петри, изображенной на рис. 23, может выглядеть следующим образом:

```
environment_random_type::srandom();
environment_random_type env;

// объекты-позиции
place_type p1, p2, p3, p4, p5, p6;
// объекты-переходы
transition_1_type t1;
transition_2_type t2;
transition_3_type t3;

// заполнение содержимого сети Петри
petrinet_type::content_type content;
// внесение позиций
content.add_place(p1);
content.add_place(p2);
content.add_place(p3);
content.add_place(p4);
content.add_place(p5);
content.add_place(p6);
// внесение переходов
content.add_transition(t1);
content.add_transition(t2);
content.add_transition(t3);
// внесение входных дуг
content.add_arc(p1, t1);
content.add_arc(p2, t2);
content.add_arc(p3, t2);
content.add_arc(p3, t3);
content.add_arc(p4, t3);
// внесение выходных дуг
content.add_arc(t1, p3);
content.add_arc(t1, p4);
content.add_arc(t2, p5);
content.add_arc(t3, p5);
content.add_arc(t3, p6);
// помещение фишек в позиции
```

```

content.add_token(p1, 2);
content.add_token(p2);
// создание сети Петри
petrinet_type petrinet(content);

// выполнение сети Петри
petrinet.live(env);

```

Здесь были использованы типы переходов, не объявленные ранее. Они унаследованы от класса простого перехода, переопределяют обработчик активации и используются для отслеживания последовательности срабатываний:

```

class transition_1_type: public transition_simple_type
{
    void on_activate(void)
    {
        cout << "transition 1 fires" << endl;
    }
};
// ...

```

Описанные классы предназначены в общем случае для построения и выполнения иерархических сетей, поэтому приведем пример построения сети, изображенной на рис. 25:

```

class transition_stop_type: public transition_simple_type
{
    void on_activate(void)
    {
        cout << "transition stop fires" << endl;
    }
};

// ...

environment_random_type env;

// создание и заполнение составного перехода
place_type started, stopped;
transition_stop_type stop;
transition_composite_type::content_type cnt;
cnt.add_place(started);
cnt.add_place(stopped);
cnt.add_transition(stop);
cnt.add_arc(started, stop);
cnt.add_arc(stop, stopped);
cnt.add_token(started);
transition_composite_type composite(cnt);

// создание и заполнение сети с составным переходом
petrinet_type::content_type content;
place_type begin, end;

```

```

int maxwidth = 0;
for (int i = 0; i < multilevel.size(); i++)
    maxwidth = std::max(maxwidth, (int) multilevel[i].size());
return maxwidth;
}

public:

// добавление работы
void add_job(job_abstract_type &job)
{
    // добавляемой работы не должно быть в списке
    assert(m_jobs.find(&job) == m_jobs.end());

    m_jobs.insert(jobs_type::value_type(&job, m_jobs.size()));
}

// добавление зависимости одной работы от другой
void add_dependence(
    job_abstract_type &dst,
    job_abstract_type &src)
{
    // работы должны быть различными
    assert(&src != &dst);

    // обе работы уже должны быть в списке
    assert(m_jobs.find(&dst) != m_jobs.end());
    assert(m_jobs.find(&src) != m_jobs.end());

    m_deps.insert(deps_type::value_type(m_jobs[&dst], m_jobs[&src]));
}

// выполнение всего комплекса работ
void run(void)
{
    // получим список подлежащих выполнению работ
    joblist_type joblist = get_joblist();
    // построим ярусно-параллельную структуру номеров работ
    multilevel_type multilevel = build(get_deplist());

    // выполним по очереди каждый ярус работ
    for (int l = 0; l < multilevel.size(); l++)
    {
        #pragma omp parallel for num_threads(multilevel[l].size())
        for (int i = 0; i < multilevel[l].size(); i++)
            joblist[multilevel[l][i]]->run();
    }
}
};

```

```

    // значит текущая работа остается неопределенной
    lev = -1;
    break;
}
else
    // иначе ярус на единицу больше максимального
    lev = std::max(lev, levlist[deplist[*it][j]] + 1);
};

// если ярус определили, добавляем в определенные
if (lev >= 0)
{
    levlist[*it] = lev;
    determined.push_back(*it);
};
};

// если нечего не удалось определить,
// видимо, у нас циклические зависимости
assert(determined.size() > 0);

// выкинем из неопределенных те, что определили
std::set<int> diff;
std::set_difference(
    nondetermined.begin(), nondetermined.end(),
    determined.begin(), determined.end(),
    std::inserter(diff, diff.end()));
nondetermined.swap(diff);
};

// построим ярусно-параллельную структуру
// из всех работ на основе вычисленных ярусов
multilevel_type multilevel(height(levlist));
for (int i = 0; i < deplist.size(); i++)
    multilevel[levlist[i]].push_back(i);

return multilevel;
}

// получение высоты ярусно-параллельной структуры
static int height(const std::vector<int> &level)
{
    int maxlevel = 0;
    for (int i = 0; i < level.size(); i++)
        maxlevel = std::max(maxlevel, level[i]);
    return maxlevel + 1;
}

// получение ширины ярусно-параллельной структуры
static int width(const multilevel_type &multilevel)
{

```

```

content.add_place(begin);
content.add_place(end);
content.add_transition(composite);
content.add_arc(begin, composite);
content.add_arc(composite, end);
content.add_token(begin);
petrinet_type petrinet(content);

// выполнение сети
petrinet.live(env);

```

Наконец, рассмотрим варианты решения задачи об обедающих философах. Прежде всего, рассмотрим вариант одноуровневой сети (рис. 32). Такая сеть потребует два типа переходов:

```

// переход "приступает к еде"
class transition_eating_start_type: public transition_simple_type
{
private:
    int m_num;
    void on_activate(void)
    {
        cout << "Философ " << m_num << " приступил к еде" << endl;
    }
public:
    transition_eating_start_type(int num): m_num(num) {}
};

// переход "приступает к размышлениям"
class transition_eating_stop_type: public transition_simple_type
{
private:
    int m_num;
    void on_activate(void)
    {
        cout << "Философ " << m_num << " приступил к размышлениям" << endl;
    }
public:
    transition_eating_stop_type(int num): m_num(num) {}
};

```

Оба типа принимают на вход параметр – порядковый номер философа, который будет использоваться для отображения последовательности срабатываний. Код же создания и выполнения соответствующей сети Петри может быть следующим:

```

const int N = 5;
// создаем элементы сети
vector<place_type> eating(N), thinking(N), fork(N);
vector<transition_eating_start_type> start;
vector<transition_eating_stop_type> stop;
for (int i = 0; i < N; i++)
{

```

```

start.push_back(transition_eating_start_type(i));
stop.push_back(transition_eating_stop_type(i));
};

// заполняем содержимое сети
petrinet_type::content_type content;
for (int i = 0; i < N; i++)
{
    // добавляем в сеть позиции
    content.add_place(eating[i]);
    content.add_place(thinking[i]);
    content.add_place(fork[i]);
    // добавляем в сеть переходы
    content.add_transition(start[i]);
    content.add_transition(stop[i]);
};
for (int i = 0; i < N; i++)
{
    // входные и выходные дуги перехода start
    content.add_arc(thinking[i], start[i]);
    content.add_arc(fork[i], start[i]);
    content.add_arc(fork[(i + 1) % N], start[i]);
    content.add_arc(start[i], eating[i]);
    // входные и выходные дуги перехода stop
    content.add_arc(eating[i], stop[i]);
    content.add_arc(stop[i], thinking[i]);
    content.add_arc(stop[i], fork[i]);
    content.add_arc(stop[i], fork[(i + 1) % N]);
};
for (int i = 0; i < N; i++)
{
    // кладем фишки
    content.add_token(thinking[i]);
    content.add_token(fork[i]);
};
// создаем и запускаем сеть
petrinet_type petrinet(content);
petrinet.live(env);

```

Прежде всего, осуществляется создание элементов, указателями на которые будет впоследствии манипулировать объект класса `content_type`. После этого в содержимое вносятся все позиции и переходы, затем дуги и фишки. Наконец, создается и выполняется сеть. Следует учитывать, что такая сеть будет жива всегда, поэтому, фактически, в последней строке осуществляется вечный цикл.

Теперь рассмотрим построение иерархической сети, изображенной на рис. 33. В этом случае нам нужен другой тип перехода, который мы определим следующим образом:

```

// функции получения списков работ и зависимостей
joblist_type get_joblist(void) const
{
    // заполняем список работ
    joblist_type joblist(m_jobs.size());
    jobs_type::const_iterator it;
    for (it = m_jobs.begin(); it != m_jobs.end(); it++)
        joblist[it->second] = it->first;
    return joblist;
}

deplist_type get_deplist(void) const
{
    // строим таблицу зависимостей работ
    deplist_type deplist(m_jobs.size());
    deps_type::const_iterator it;
    for (it = m_deps.begin(); it != m_deps.end(); it++)
        deplist[it->first].push_back(it->second);
    return deplist;
}

// построение ярусно-параллельной структуры
// на основе зависимостей работ между собой
static multilevel_type build(const deplist_type &deplist)
{
    // вычислим ярусы всех работ на основе зависимостей
    std::vector<int> levlist(deplist.size());

    // множество номеров работ, ярусы которых еще не определены
    std::set<int> nondetermined;
    for (int i = 0; i < levlist.size(); i++)
        nondetermined.insert(i);

    // цикл определения ярусов работ
    while (nondetermined.size() > 0)
    {
        std::vector<int> determined;
        // пройдемся по всем неопределенным еще работам
        std::set<int>::iterator it;
        for (it = nondetermined.begin(); it != nondetermined.end(); it++)
        {
            // если зависимостей нет - нулевой ярус
            int lev = 0;
            // пройдемся по всем зависимостям, если есть
            for (int j = 0; j < deplist[*it].size(); j++)
            {
                // если зависимость найдена среди неопределенных
                if (nondetermined.find(deplist[*it][j]) !=
                    nondetermined.end())
                {

```

Приложение 2. Классы построения и выполнения комплекса работ

Приводятся классы, используемые во второй главе и реализующие построение и выполнение комплекса работ на основе переданного списка работ и зависимостей между ними. Текущий вариант представляет собой последовательную версию, распараллеленную с использованием директив OpenMP.

```
// абстрактный интерфейс работы
class job_abstract_type
{
public:

    // выполнение работы
    // получение исходных данных и вывод результата
    // выполняются внутри через разделяемые ресурсы
    virtual void run(void) = 0;

};

// комплекс работ
class jobcomplex_type: public job_abstract_type
{
private:

    // список работ в порядке добавления
    typedef std::vector<job_abstract_type *> joblist_type;

    // таблица зависимостей работ по номерам
    typedef std::vector<std::vector<int> > deplist_type;

    // номера работ в ярусно-параллельной форме
    typedef std::vector<std::vector<int> > multilevel_type;

    // =====
    // описание содержимого комплекса работ
    // =====

    // множество работ с отображением на их номера
    typedef std::map<job_abstract_type *, int> jobs_type;
    jobs_type m_jobs;

    // множество зависимостей работ
    typedef std::set<std::pair<int, int> > deps_type;
    deps_type m_deps;

private:
```

```
// составной переход "философ ест"
class transition_eating_type: public transition_composite_type
{
private:
    int m_num;

    void on_activate(void)
    {
        cout << "Философ " << m_num << " приступил к еде" << endl;
    }
    void on_passivate(void)
    {
        cout << "Философ " << m_num << " приступил к размышлениям" << endl;
    }

public:
    transition_eating_type(const content_type &content, int num):
        transition_composite_type(content),
        m_num(num)
    {}
};
```

Обработчики этого перехода снова используются для отслеживания последовательности срабатываний, однако на этот раз срабатывают начало и завершение составного перехода. Каждый составной переход содержит вложенную сеть в соответствии с рис. 25. Ниже следует код, реализующий создание и выполнение такой сети:

```
const int N = 5;
// создаем содержимое составных переходов
vector<place_type> started(N), stopped(N);
vector<transition_simple_type> stop(N);
// и сами составные переходы
vector<transition_eating_type> eating;
for (int i = 0; i < N; i++)
{
    transition_composite_type::content_type cnt;
    cnt.add_place(started[i]);
    cnt.add_place(stopped[i]);
    cnt.add_transition(stop[i]);
    cnt.add_arc(started[i], stop[i]);
    cnt.add_arc(stop[i], stopped[i]);
    cnt.add_token(started[i]);
    eating.push_back(transition_eating_type(cnt, i));
};

// заполняем содержимое сети
vector<place_type> thinking(N), fork(N);
petrinet_type::content_type content;
for (int i = 0; i < N; i++)
{
```

```

// добавляем в сеть позиции
content.add_place(thinking[i]);
content.add_place(fork[i]);
// добавляем в сеть переходы
content.add_transition(eating[i]);
};
for (int i = 0; i < N; i++)
{
    // входные дуги перехода eating
    content.add_arc(thinking[i], eating[i]);
    content.add_arc(fork[i], eating[i]);
    content.add_arc(fork[(i + 1) % N], eating[i]);
    // выходные дуги перехода eating
    content.add_arc(eating[i], thinking[i]);
    content.add_arc(eating[i], fork[i]);
    content.add_arc(eating[i], fork[(i + 1) % N]);
};
for (int i = 0; i < N; i++)
{
    // кладем фишки
    content.add_token(thinking[i]);
    content.add_token(fork[i]);
};
// создаем и запускаем сеть
petrinet_type petrinet(content);
petrinet.live(env);

```

Первым делом создаются и заполняются составные переходы, после чего они используются при включении элементов в содержимое внешней сети. До текущего момента мы не затрагивали вопрос параллельного выполнения, и в данном случае длительные переходы работают параллельно лишь теоретически. Далее же мы рассмотрим возможности реального распараллеливания выполнения длительных переходов.

4.2.2 Выполнение параллельных процессов

Для реализации параллельных программ мы будем отождествлять длительные подзадачи с составными переходами, аналогичными изображенному на рис. 25. Предложенный ранее вариант окружения, в котором ожидание срабатывания основывалось на случайном выборе перехода, в данном случае уже не подходит по следующей причине. Каждый составной переход содержит один простой переход, срабатывающий не произвольно, а вследствие завершения выполнения соответствующей подзадачи. При этом выполнение длительной подзадачи вынесено в отдельный

```

    return mtx;
}
};

// вектор - матрица в один столбец
template <class e_t>
class vector_type: public matrix_type<e_t>
{
public:

    typedef matrix_type<e_t> base_type;
    typedef typename base_type::element_type element_type;

public:

    // конструктор
    vector_type(int vsize):
        base_type(vsize, 1)
    {
    }

    // конструктор - преобразование типа
    vector_type(const base_type &src):
        base_type(src.vsize(), 1)
    {
        assert(src.hsize() == 1);

        this->operator =(src);
    }

    // обращение к элементам по индексу (нумерация с единицы)
    const element_type & operator ()(int i) const
    {
        return static_cast<const base_type *>(this)->operator ()(i, 1);
    }

    element_type & operator ()(int i)
    {
        return static_cast<base_type *>(this)->operator ()(i, 1);
    }

    // присваивание
    vector_type & operator =(const base_type &src)
    {
        return static_cast<vector_type &>(
            static_cast<base_type *>(this)->operator =(src));
    }
};

```



```

#pragma omp parallel for
for (int i = 1; i <= vsize(); i++)
    for (int j = 1; j <= hsize(); j++)
        (*this)(i, j) -= src(i, j);

return *this;
}

// сумма двух матриц
friend
matrix_type operator +(
    const matrix_type &src1, const matrix_type &src2)
{
    assert(
        src1.hsize() == src2.hsize() && src1.vsize() == src2.vsize());

    matrix_type mtx = src1;
    return (mtx += src2);
}

// разность двух матриц
friend
matrix_type operator -(
    const matrix_type &src1, const matrix_type &src2)
{
    assert(
        src1.hsize() == src2.hsize() && src1.vsize() == src2.vsize());

    matrix_type mtx = src1;
    return (mtx -= src2);
}

// перемножение двух матриц
friend
matrix_type operator *(
    const matrix_type &src1, const matrix_type &src2)
{
    assert(src1.hsize() == src2.vsize());

    matrix_type mtx(src1.vsize(), src2.hsize());

#pragma omp parallel for
for (int i = 1; i <= mtx.vsize(); i++)
{
    for (int j = 1; j <= mtx.hsize(); j++)
    {
        element_type sum(0);
        for (int k = 1; k <= src1.hsize(); k++)
            sum += src1(i, k) * src2(k, j);
        mtx(i, j) = sum;
    };
};
};

```

параллельный ресурс.

Интерфейсы OpenMP и MPI не обеспечивают необходимой гибкости для реализации сетей Петри, поэтому обратимся к низкоуровневым средствам распараллеливания. Для примера мы рассмотрим многопоточное распараллеливание, хотя с использованием сетевых коммуникаций может быть выполнено и распараллеливание в системах с распределенной памятью.

Выполним распараллеливание выполнения длительных переходов с помощью интерфейса Win32 API. Для этого создадим следующий класс окружения:

```

// класс окружения с поддержкой многопоточного выполнения
class environment_type: public
    petrinet_type::environment_abstract_type
{
public:
    // интерфейс работы, выполняемой во время длительного перехода
    class longjob_abstract_type
    {
    public:
        // функция выполнения работы
        virtual void run(void) = 0;
    };

    // длительный переход, характеризующий выполнение работы
    class transition_long_type: public transition_composite_type
    {
    private:
        // окружение
        environment_type *m_penv;
        // номер перехода в списке длительных переходов окружения
        int m_index;

        // вызовы инициализации и финализации выполнения работы окружением
        void on_activate(void)
        {
            m_penv->initialize_longjob(m_index);
        }
        void on_passivate(void)
        {
            m_penv->finalize_longjob(m_index);
        }
    public:
        transition_long_type(
            const content_type &content, environment_type *penv, int index):
            transition_composite_type(content), m_penv(penv), m_index(index)
        {}
    };

private:

```

```

// структура данных, связанная с каждым длительным переходом
struct longjob_data_type
{
    // содержимое соответствующего длительного перехода
    place_type started, stopped;
    transition_simple_type stop;
    // связанный поток
    HANDLE thread;
    // указатель на соответствующую работу (параметр потока)
    longjob_abstract_type *pjob;
};

// отображение номеров длительных переходов на связанные структуры
// в виде вектора хранить нельзя, поскольку тогда при добавлении
// нового элемента становятся невалидными указатели на старые
std::map<int, longjob_data_type> m_ljobdata;
// отображение переходов завершения (stop)
// на номера соответствующих длительных переходов
std::map<transition_abstract_type *, int> m_stormap;

// функция потока - выполняет вызов функции выполнения работы
static DWORD WINAPI thr_proc(LPVOID param)
{
    // выполнение работы
    static_cast<longjob_abstract_type *>(param)->run();
    return 0;
}

public:
    transition_long_type allocate_long_transition(
        longjob_abstract_type &longjob)
    {
        // номер создаваемого длительного перехода
        int index = m_ljobdata.size();

        // разместим данные длительного перехода
        longjob_data_type &ljdata = m_ljobdata[index];
        ljdata.pjob = &longjob;

        // добавим переход завершения в отображение для поиска номера
        m_stormap[&ljdata.stop] = index;

        // создадим и заполним содержимое длительного перехода
        transition_composite_type::content_type content;
        content.add_place(ljdata.started);
        content.add_place(ljdata.stopped);
        content.add_transition(ljdata.stop);
        content.add_arc(ljdata.started, ljdata.stop);
        content.add_arc(ljdata.stop, ljdata.stopped);
        content.add_token(ljdata.started);
        return transition_long_type(content, this, index);
    }
    void initialize_longjob(int i)

```

```

int hsize(void) const
{
    return m_hsize;
}

// обращение к элементам по индексам (нумерация с единицы)
const element_type & operator ()(int i, int j) const
{
    assert(i > 0 && j > 0 && i <= vsize() && j <= hsize());

    return m_data[(i - 1) * hsize() + j - 1];
}

element_type & operator ()(int i, int j)
{
    return const_cast<element_type &>(
        static_cast<const matrix_type *>(this)->operator ()(i, j));
}

// присваивание матриц одинаковых размеров
matrix_type & operator =(const matrix_type &src)
{
    assert(vsize() == src.vsize() && hsize() == src.hsize());

    memcpy(m_data, src.m_data,
        m_vsize * m_hsize * sizeof(element_type));

    return *this;
}

// прибавление матрицы
matrix_type & operator +=(const matrix_type &src)
{
    assert(hsize() == src.hsize() && vsize() == src.vsize());

    #pragma omp parallel for
    for (int i = 1; i <= vsize(); i++)
        for (int j = 1; j <= hsize(); j++)
            (*this)(i, j) += src(i, j);

    return *this;
}

// вычитание матрицы
matrix_type & operator -=(const matrix_type &src)
{
    assert(hsize() == src.hsize() && vsize() == src.vsize());

```

Приложение 1. Шаблоны классов матрицы и вектора

Приводится пример реализации шаблона класса матрицы, предоставляющего интерфейс выполнения основных операций, а также унаследованный от него шаблон вектора, являющийся матрицей шириной в один столбец.

```
// матрица
template <class e_t>
class matrix_type
{
public:

    typedef e_t element_type;

private:

    int m_vsize, m_hsize;
    element_type *m_data;

public:

    // конструктор
    matrix_type(int vsize, int hsize):
        m_vsize(vsize), m_hsize(hsize)
    {
        assert(m_vsize > 0 && m_hsize > 0);

        m_data = new element_type[m_vsize * m_hsize];
    }

    // конструктор копирования
    matrix_type(const matrix_type &src):
        m_vsize(src.vsize()), m_hsize(src.hsize())
    {
        m_data = new element_type[m_vsize * m_hsize];
        this->operator =(src);
    }

    // деструктор
    ~matrix_type(void)
    {
        delete[] m_data;
    }

    // размеры по вертикали и горизонтали
    int vsize(void) const
    {
        return m_vsize;
    }
}
```

```
{
    // инициализация работы - создание потока
    DWORD dwId;
    m_ljobdata[i].thread = ::CreateThread(
        NULL, 0,
        thr_proc, m_ljobdata[i].pjob,
        0, &dwId);
    assert(m_ljobdata[i].thread != NULL);
}

void finalize_longjob(int i)
{
    // завершительные действия - освобождение ресурсов
    // к этому моменту поток уже закончил работу
    ::CloseHandle(m_ljobdata[i].thread);
}

public:
    // инициализация датчика псевдослучайных чисел
    static void srandom(void)
    {
        srand((unsigned) time(NULL));
    }

    // генерация псевдослучайного числа в интервале [0, bound)
    static int random(int bound)
    {
        assert(bound > 0);
        return int(rand() / (RAND_MAX + 1.0) * bound);
    }

    // ожидание срабатывания перехода
    int wait(const petrinet_type::enabledlist_type &enabled)
    {
        // списки позиций переданных переходов
        // busy - переходы завершения работ, которые еще не завершены
        // finished - переходы завершения работ, которые уже завершены
        // free - переходы, не являющиеся переходами завершения работ
        std::vector<int> busy, finished, free;
        // хэндлы потоков, связанных с незавершенными работами
        std::vector<HANDLE> hdl;
        // проходим по всему списку переходов
        int pos = 0;
        petrinet_type::enabledlist_type::const_iterator it;
        for (it = enabled.begin(); it != enabled.end(); it++)
        {
            // если это не переход завершения работы, кладем в свободные
            if (m_stopmap.find(*it) == m_stopmap.end())
                free.push_back(pos);
            else
            {
                // иначе получаем хэндл связанного потока
                HANDLE h = m_ljobdata[m_stopmap[*it]].thread;
                // и проверяем, завершился ли он
                if (::WaitForSingleObject(h, 0) == WAIT_OBJECT_0)
                    finished.push_back(pos);
            }
        }
    }
}
```

```

else
{
    // если не завершился, добавляем хэндл на ожидание
    busy.push_back(pos);
    hdl.push_back(h);
};
};
pos++;
};
// формируем код возврата
int rc;
// если есть завершившиеся работы, возвращаем
// позицию одного из соответствующих переходов
if (finished.size() > 0)
    rc = finished[random(finished.size())];
// если есть свободные переходы, вернем позицию одного из них
else if (free.size() > 0)
    rc = free[random(free.size())];
else
{
    // в остальных случаях дождемся завершения одного из потоков
    DWORD dw = ::WaitForMultipleObjects(
        hdl.size(), &hdl[0], FALSE, INFINITE);
    assert(dw >= WAIT_OBJECT_0 && dw < WAIT_OBJECT_0 + hdl.size());
    // и вернем позицию соответствующего перехода
    rc = busy[dw - WAIT_OBJECT_0];
};
return rc;
}
};

```

Приведенный код куда более громоздок по сравнению с приведенным ранее классом `environment_random_type`. Прежде всего, объявляются используемые классом окружения типы. Среди них интерфейс длительной работы, выполняемой в рамках одного длительного перехода, и класс длительного перехода. Последний унаследован от класса составного перехода и определяет обработчики событий активации и деактивации, которые иницируют начало и завершение выполнения длительной работы в параллельном потоке. Эти операции непосредственно реализует объект окружения, класс же длительного перехода передает ему необходимые для их иницирования данные.

С каждым длительным переходом сети ассоциирована структура данных `longjob_data_type`. Она содержит объявления внутренних элементов составного перехода, а также данные, необходимые для выполнения длительной работы в отдельном потоке. Заполнение такой структуры и добавление ее к контейнеру

пользовании для параллельного программирования принципиально иных средств, не являющихся модификациями или надстройками существующих популярных языков. Одним из возможных направлений является декларативное программирование, что отчасти и было нами продемонстрировано, поскольку приведенные программы, хоть и были реализованы на императивном языке, зачастую по структуре представлялись декларативно.

Во-вторых, была сделана попытка проиллюстрировать следующий факт, вытекающий из рассмотрения тех же обстоятельств с обратной стороны. Несмотря на указанные сложности, зачастую, все же, императивные языки могут быть использованы для реализации параллельных алгоритмов. При этом может быть реализована некая абстрактная машина, принимающая на входе набор декларативно описывающих алгоритм данных, которая инкапсулирует внутри себя механизмы параллельного выполнения. Программный интерфейс такой машины принципиально не зависит от средств распараллеливания, с помощью которых она реализована и которые в общем случае могут быть совершенно разными. Такой подход позволит не уходить от имеющейся на сегодняшней момент ситуации преобладания императивных языков и, тем самым, в какой-то степени упростит и сгладит освоение параллельного программирования.

Оба этих момента взаимно дополняют друг друга. Учитывая отсутствие на сегодняшний момент развитых и обладающих широкой популярностью средств декларативного параллельного программирования, в практических целях может быть рассмотрен подход с использованием механизма обработки декларативных данных, реализованного на императивном языке. Однако в целом такие меры являются вынужденными и требуют дополнительных затрат, вследствие чего в будущем требуется появление новых либо популяризация существующих более мощных средств разработки, изначально ориентированных на параллельное программирование и не являющихся видоизменением каких-либо средств последовательного программирования.

Заключение

В связи с неотвратно приближающимся достижением современными компьютерами своего предела тактовых частот параллельное программирование постепенно перестает быть узкоспециализированной дисциплиной для высокопроизводительных вычислений и приобретает все большую актуальность. Вследствие этого появляется немало публикаций, посвященных теоретическим вопросам распараллеливания последовательных программ, а также немало учебных пособий и статей, посвященных рассмотрению существующего инструментария. Помимо этого уже довольно давно создано немало моделей, успешно описывающих параллельное выполнение процессов и являющихся основой многих реальных разработок и некоторых малоизвестных инструментов организации параллельного выполнения программ. Таким моделям посвящено немало учебного материала теоретического характера, однако ощущается некоторый дефицит в области практического их рассмотрения с доступными примерами программной реализации с использованием современных средств. В результате рассмотрения таких моделей лишь с теоретической точки зрения они, порой, остаются вне практического арсенала программиста, в связи с чем, в силу консерватизма мышления многих программистов, возможность использования таких моделей зачастую даже не рассматривается.

Представленное описание некоторых возможных подходов к параллельному программированию, помимо очевидных целей любого учебного пособия, имеет также целью показать следующие два взаимосвязанных момента.

Во-первых, стояла задача показать проблему современного параллельного программирования, заключающуюся, прежде всего, в том, что наиболее популярные сегодня средства программирования приспособлены к описанию методов решения стоящей задачи в соответствии с образом мышления программиста, а именно в последовательной форме. Отсюда вытекает принципиальная сложность наглядного для человека представления с использованием таких средств параллельных алгоритмов и программ. В связи с этим встает ребром вопрос необходимости в ис-

m_ljobdata осуществляется вместе с созданием и заполнением соответствующего составного перехода функцией `allocate_long_transition` по инициативе вызывающего кода. Она же добавляет соответствующий завершению длительной операции простой переход в свой внутренний список, который будет использоваться в дальнейшем для идентификации длительного перехода.

Функция `initialize_longjob` создает новый поток на основе функции `thr_proc`, передавая ему в качестве параметра указатель на требующую выполнения работу. Соответственно, функция `finalize_longjob` освобождает ресурсы после завершения потока.

Наконец, функция ожидания срабатывания перехода, прежде всего, анализирует весь список переданных переходов и делит их на две группы – переходы, которые не попадают во множество переходов завершения длительных операций, и, соответственно, попадающие в него. Для каждого перехода завершения определяется соответствующий номер длительной операции и факт завершения соответствующего потока. По признаку завершения потока эти переходы также делятся на две группы. После разделения на три группы осуществляется выбор срабатывающего перехода.

Первым делом проверяется множество переходов завершения, для которых соответствующие потоки уже завершены. Если оно не пусто, возвращается произвольный номер из этого множества, поскольку такие переходы должны обрабатываться в первую очередь. В противном случае проверяется множество переходов, не являющихся переходами завершения длительных операций, поскольку они в нашем случае не требуют ожидания. Если оно не пусто, из него возвращается произвольный номер. Наконец, если все разрешенные переходы – переходы завершения, потоки которых до сих пор не завершены, выполняется ожидание завершения любого из них, после чего возвращается соответствующий номер перехода.

При такой реализации функции ожидания практически все время тратится на ожидание завершения параллельных потоков, выполняющих длительные работы. В случае же, когда переход

характеризует простые операции, к примеру, объекты синхронизации, работа сети продолжается без ожидания, вследствие чего построенная таким образом параллельная программа может быть достаточно эффективной.

Следуя принятому нами ранее подходу, реализуем код окружения с аналогичной функциональностью с помощью альтернативного программного интерфейса – POSIX Threads. Создание и уничтожение потоков происходит схожим образом. Основная возникающая сложность заключается в том, что в этом интерфейсе нет простой возможности ожидания завершения любого из нескольких потоков, вследствие чего такая функциональность будет реализована нами вручную. Для этого в классе окружения добавим два объекта синхронизации:

```
class environment_type: public
petrinet_type::environment_abstract_type
{
    // ...
private:
    // объекты mutex и cond для ожидания завершения потоков
    pthread_mutex_t m_mutex;
    pthread_cond_t m_cond;
    // номер последнего завершившегося потока
    int m_lastindex;

    // проверка кода возврата функций pthread_xxx
    static inline void chkzero(int retcode) { assert(retcode == 0); }

    // ...
};
```

Здесь также приведено объявление функции проверки кода возврата для всех функций pthreads, а также переменной, содержащей номер последнего завершившегося длительного перехода. Эта переменная нам требуется для обеспечения функциональности, аналогичной возврату значения при ожидании завершения одного из потоков.

Использование объектов синхронизации требует их инициализации и уничтожения, для чего классу окружения потребуется конструктор и деструктор:

```
class environment_type: public
petrinet_type::environment_abstract_type
{
    // ...
```

```
// формируем длительные переходы на выполнение работ
vector<environment_type::transition_long_type> eating;
for (int i = 0; i < N; i++)
    eating.push_back(env.allocate_long_transition(eatingjob[i]));

// заполняем содержимое сети
vector<place_type> thinking(N), fork(N);
petrinet_type::content_type content;

for (int i = 0; i < N; i++)
{
    // добавляем в сеть позиции
    content.add_place(thinking[i]);
    content.add_place(fork[i]);

    // добавляем в сеть переходы
    content.add_transition(eating[i]);
};

for (int i = 0; i < N; i++)
{
    // входные дуги перехода eating
    content.add_arc(thinking[i], eating[i]);
    content.add_arc(fork[i], eating[i]);
    content.add_arc(fork[(i + 1) % N], eating[i]);

    // выходные дуги перехода eating
    content.add_arc(eating[i], thinking[i]);
    content.add_arc(eating[i], fork[i]);
    content.add_arc(eating[i], fork[(i + 1) % N]);
};

for (int i = 0; i < N; i++)
{
    // кладем фишки
    content.add_token(thinking[i]);
    content.add_token(fork[i]);
};

// создаем и запускаем сеть
petrinet_type petrinet(content);
petrinet.live(env);
```

В рассмотренном фрагменте осуществляется создание длительных работ, после чего с помощью окружения создаются соответствующие составные переходы, которые позже вносятся в содержимое сети. В остальном же вызывающий код полностью идентичен рассмотренному ранее коду с использованием составных переходов.

нализировать объект `m_cond`. Перед возвратом из функции `pthread_cond_wait` объект `m_mutex` снова блокируется, в конце же функции `wait` блокировка снимается.

Реализуем на основе описанного окружения задачу с параллельным выполнением длительных операций. Не уходя далеко за примерами, рассмотрим снова задачу об обедающих философах. Будем считать, что каждому философу во время однократного приема пищи требуется произвести вычисления произвольной длительности. Отождествим длительную операцию с одним подходом к еде, как изображено на рис. 33. Для реализации этой операции создадим соответствующий класс длительной работы:

```
// длительная операция "философ ест"
class longjob_eating_type:
public environment_type::longjob_abstract_type
{
private:
    int m_num;

    void run(void)
    {
        cout << "Философ " << m_num << " приступил к еде" << endl;

        // ... какие-либо длительные вычисления

        cout << "Философ " << m_num << " приступил к размышлениям" << endl;
    }

public:
    longjob_eating_type(int num): m_num(num) {}
};
```

Код, реализующий создание и выполнение сети с параллельным выполнением таких работ, отличается от приведенного ранее кода на основе составных переходов лишь в части, касающейся создания этих переходов:

```
environment_type env;
const int N = 5;

// формируем список работ
vector<longjob_eating_type> eatingjob;
for (int i = 0; i < N; i++)
    eatingjob.push_back(longjob_eating_type(i));
```

```
public:
    // вместе с объектом окружения создаются и уничтожаются
    // объекты синхронизации mutex и cond
    environment_type(void)
    {
        chkzero(::pthread_mutex_init(&m_mutex, NULL));
        chkzero(::pthread_cond_init(&m_cond, NULL));
    }
    ~environment_type(void)
    {
        chkzero(::pthread_cond_destroy(&m_cond));
        chkzero(::pthread_mutex_destroy(&m_mutex));
    }
    // ...
};
```

Для взаимодействия функции потока с объектами синхронизации ей на этот раз передается не только указатель на выполняемую работу, а следующая структура данных `thr_param`:

```
struct longjob_data_type
{
    // ...
    // связанный поток
    pthread_t thread;
    // структура параметров потока
    struct thr_param
    {
        // указатель на соответствующую работу
        longjob_abstract_type *pjob;
        // окружение
        environment_type *penv;
        // номер в списке длительных переходов окружения
        int index;
        // признак завершения выполнения
        bool finished;
    } param;
};
```

Соответственно, в функции `allocate_long_transition` потребуется инициализация этой структуры:

```
transition_long_type allocate_long_transition(longjob_abstract_type
&longjob)
{
    // ...
    // разместим данные длительного перехода
    longjob_data_type &ljdata = m_ljobdata[index];
    ljdata.param.pjob = &longjob;
    ljdata.param.penv = this;
    ljdata.param.index = index;

    // добавим переход завершения в отображение для поиска номера
    // ...
}
```

```
}
```

В соответствии с используемым программным интерфейсом изменяются функции инициализации выполнения работы и последующего освобождения ресурсов:

```
void initialize_longjob(int i)
{
    // установка признака - поток не завершен
    m_ljobdata[i].param.finished = false;
    // инициализация работы - создание потока
    chkzero(::pthread_create(
        &m_ljobdata[i].thread, NULL,
        thr_proc, &m_ljobdata[i].param));
}
void finalize_longjob(int i)
{
    // завершительные действия - освобождение ресурсов
    // к этому моменту поток уже закончил работу
    chkzero(::pthread_join(m_ljobdata[i].thread, NULL));
}
```

Изменению подлежит и функция потока, которая теперь помимо выполнения работы должна сообщить функции ожидания о факте завершения, при этом выложив номер соответствующего длительного перехода:

```
static void *thr_proc(void *param)
{
    longjob_data_type::thr_param *p =
        static_cast<longjob_data_type::thr_param *>(param);
    // выполнение работы
    p->rjob->run();
    // установка блокировки
    chkzero(::pthread_mutex_lock(&p->penv->m_mutex));
    // модификация признаков завершения потока
    p->penv->m_lastindex = p->index;
    p->finished = true;
    // отправка сигнала о завершении потока
    chkzero(::pthread_cond_signal(&p->penv->m_cond));
    // снятие блокировки
    chkzero(::pthread_mutex_unlock(&p->penv->m_mutex));
    return NULL;
}
```

Наконец, меняется функция ожидания срабатывания перехода:

```
int wait(const petrinet_type::enabledlist_type &enabled)
{
    chkzero(::pthread_mutex_lock(&m_mutex));
    // списки позиций переданных переходов
    // finished - переходы завершения работ, которые уже завершены
    // free - переходы, не являющиеся переходами завершения работ
```

```
std::vector<int> finished, free;
// отображение номеров длительных переходов на позиции
// переходов завершения работ, которые еще не завершены
std::map<int, int> busybyindex;
// проходим по всему списку переходов
int pos = 0;
petrinet_type::enabledlist_type::const_iterator it;
for (it = enabled.begin(); it != enabled.end(); it++)
{
    // если это не переход завершения работы, кладем в свободные
    if (m_stopmap.find(*it) == m_stopmap.end())
        free.push_back(pos);
    else
    {
        // иначе проверяем, завершился ли поток
        if (m_ljobdata[m_stopmap[*it]].param.finished)
            finished.push_back(pos);
        else
            // если не завершился, добавляем информацию для ожидания
            busybyindex[m_stopmap[*it]] = pos;
    };
    pos++;
};
// формируем код возврата
int rc;
// если есть завершившиеся работы, возвращаем
// позицию одного из соответствующих переходов
if (finished.size() > 0)
    rc = finished[random(finished.size())];
// если есть свободные переходы, вернем позицию одного из них
else if (free.size() > 0)
    rc = free[random(free.size())];
else
{
    // в остальных случаяхждемся завершения одного из потоков
    chkzero(::pthread_cond_wait(&m_cond, &m_mutex));
    // и вернем позицию соответствующего перехода
    rc = busybyindex[m_lastindex];
};
chkzero(::pthread_mutex_unlock(&m_mutex));
return rc;
}
```

В начале функции выполняется блокировка объекта `m_mutex`, по причине чего становится возможной проверка данных длительных переходов в части информации об их завершении. При выполнении же ожидания внутри функции `pthread_cond_wait` объект `m_mutex` временно освобождается, в результате чего завершившийся поток сможет заблокировать его, передать номер соответствующего длительного перехода и сиг-