МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ АГЕНТСТВО ОБРАЗОВАНИЯ

Государственное образовательное учреждение
высшего профессионального образования
“Оренбургский государственный университет”

Е.Н. ИШАКОВА

ТЕОРИЯ ВЫЧИСЛИТЕЛЬНЫХ ПРОЦЕССОВ

Рекомендовано Ученым советом государственного образовательного учреждения высшего профессионального образования «Оренбургский государственный университет» в качестве учебного пособия для студентов, обучающихся по программам высшего профессионального образования по специальности «Программное обеспечение вычислительной техники и автоматизированных систем»

Оренбург 2007

УДК 004(075.8)
ББК 32.81я73
И 97

Рецензент
доктор технических наук, профессор Пищухин А.М.

Ишакова Е.Н.
И 97 **Теория вычислительных процессов: учебное пособие / Е.Н. Ишакова. – Оренбург: ГОУ ОГУ, 2007. – 160 с.**

ISBN

В пособии рассмотрены основы теоретического программирования: теория схем программ, семантическая теория программ, теория параллельных вычислений, теория протоколов и интерфейсов. Теоретический материал дополнен примерами и задачами с решениями.

Учебное пособие предназначено для студентов, обучающихся по программам высшего профессионального образования по специальности 230105.65 «Программное обеспечение вычислительной техники и автоматизированных систем», при изучении дисциплины «Теория вычислительных процессов».

И $\frac{140400000}{6Л9-07}$ ББК 32.81я73

ISBN

Содержание

Введение

В данном учебном пособии рассматриваются основы теоретического программирования. Следуя А.П. Ершову, мы употребляем термин «теоретическое программирование» в качестве названия математической дисциплины, изучающей синтаксические и семантические свойства программ, их структуру, преобразования, процесс их составления и исполнения. Эта теоретическая научная дисциплина изучает фундаментальные понятия и законы основной науки и на основании обнаруженных закономерностей строит более частные математические модели исследуемых объектов, на которых ставит и решает прикладные задачи. В нашем случае объект моделирования – программа – уже представляет собой абстрактный объект.

Настоящее пособие затрагивает следующие основные направления исследований теоретического программирования.

1 Математические основы программирования. Основная цель исследований – развитие математического аппарата, ориентированного на теоретическое программирование, разработка общей теории машинных вычислений. Эта теория тесно соприкасается с теорией алгоритмов и вычислимых функций, теорией автоматов и формальных языков, логикой, алгеброй, с теорией сложности вычислений.

2 Теория схем программ. В этих работах внимание концентрируется на изучении структурных свойств и преобразований программ, а именно тех, которые отличают программы от других способов задания алгоритмов. Главным объектом исследования становится схема программы – математическая модель программы, в которой с той или иной степенью детализации отражено строение программы, взаимодействие составляющих ее компонентов.

3 Семантическая теория программ. Семантика программы или отдельных конструкций языков программирования – это их смысл, математический смысл для программиста и описание функционирования для машины. Этот раздел теоретического программирования изучает методы формального описания семантики программ, семантические методы преобразования и доказательства утверждений о программах. В частности, работы по методам проверки семантической правильности программ нацелены на автоматизацию их отладки и автоматический синтез программ.

4 Теория параллельных вычислений. Исследования в этой области направлены на разработку и обоснование новых методов программирования, прежде всего методов программирования параллельных процессов. В частности, изучаются модели, структуры и функционирование операционных систем, методы распараллеливания алгоритмов и программ, ведется поиск новых архитектурных принципов конструирования вычислительных машин и систем на основе результатов и рекомендаций теоретического программирования и вычислительной математики.

Теоретический материал пособия дополнен примерами и задачами с решениями, а также вопросами и заданиями для самоконтроля, тестами проверки усвоения материала.

1 Теория схем программ

1.1 Программы и схемы программ

Определение 1.1. Схемы программ - это математические модели программ, отвечающие требованиям:

- позволяют изучать свойства достаточно широких классов программ, а не отдельных конкретных программ;
- сохраняют все интересующие исследователя свойства и особенности рассматриваемого класса программ;
- позволяют игнорировать несущественные для данной проблемы свойства, например, синтаксические детали;
- модифицируемы и допускают нововведения, отслеживающие развитие языков программирования;
- дают возможность отделить разрешимые и неразрешимые свойства программ и классов программ;
- удобны, если структура модели изобразительно подобна структуре программ, что дает возможность привлекать на некоторых этапах исследований программистскую интуицию /1, 2/.

Главное отличие схем от программ состоит в следующем. Арифметические, логические и другие выражения программы образуются с помощью символов операций и указателей функций, семантика которых фиксирована языком программирования (для базовых операций) или выводима по определенным правилам композиции из семантики базовых операций. Поэтому каждый оператор программы единственным образом задает некоторую функцию над данными, а из этих функций образуется (вычислимая) функция, задаваемая программой в целом. В схемах программ индивидуальные операции и функции заменены абстрактными функциональными и предикатными символами. Если вместо каждого такого символа подставить конкретную функцию или предикат, то схема превращается в программу. Схема, в отличие от программы, не задает алгоритма вычисления, она описывает строение множества программ, получающихся из нее в результате задания различных интерпретаций.

1.2 Стандартные схемы программ

1.2.1 Базис класса стандартных схем программ

Стандартные схемы программ (ССП) характеризуются базисом и структурой схемы. Базис класса фиксирует символы, из которых строятся схемы, указывает их роль (переменные, функциональные символы и др.), задает вид выражений и операторов схем.

Определение 1.2. Полный базис ß класса стандартных схем состоит из четырех непересекающихся счетных множеств символов и множества опера-

торов - слов, построенных из этих символов. Множества символов полного базиса имеют вид:

1) $X = (x, x_1,\ldots, y, y_1, \ldots, z, z_1,\ldots\}$ - множество символов, называемых переменными;

2) $F = \{f^{(0)}, f^{(1)}, f^{(2)},\ldots, g^{(0)}, g^{(1)}, g^{(2)},\ldots, h^{(0)}, h^{(1)}, h^{(2)},\ldots\}$ - множество функциональных символов; верхний индекс задает местность символа; нульместные символы $\{f^{(0)}, g^{(0)}, h^{(0)},\ldots\}$ называют константами и обозначают начальными буквами латинского алфавита $a, b, c,\ldots$;

3) $P = \{p^{(0)}, p^{(1)}, p^{(2)},\ldots, q^{(0)}, q^{(1)}, q^{(2)},\ldots\}$ - множество предикатных символов; верхний индекс задает местность символа; нульместные предикатные символы $\{p^{(0)}, q^{(0)},\ldots\}$ называют логическими константами;

4) (*старт*, *стоп*, (,), , , := } - множество специальных символов.

Определение 1.3. Термами (функциональными выражениями) называются слова, построенные из переменных, функциональных и специальных символов по следующим правилам:

1) односимвольные слова, состоящие из переменных или констант, являются термами;

2) слово τ вида $f^{(n)}(\tau_1, \tau_2,..., \tau_n)$, где $\tau_1, \tau_2,..., \tau_n$ - термы, является термом;

3) те и только те слова, о которых говорится в пунктах 1 и 2, являются термами.

Пример 1.1. Термами являются: $x, f^{(0)}, a, f^{(1)}(x), g^{(2)}(x, h^{(3)}(y, a, x))$.

Определение 1.4. Тестами (логическими выражениями) называются логические константы и слова вида $p^{(n)}(\tau_1, \tau_2,..., \tau_n)$.

Пример 1.2. Тестами являются: $p^{(0)}, p^{(1)}(x), g^{(3)}(x, y, a), p^{(1)}(f^{(2)}(x, y))$.

Допускается в функциональных и логических выражениях опускать индексы местности, если это не приводит к двусмысленности или противоречию.

Множество операторов состоит из операторов пяти типов:

1) начальный оператор - слово вида *старт*$(x_1,\ldots, x_к)$, где $k\geq 0$, а $x_1,\ldots, x_k$ - попарно различные переменные из базиса ß, называемые результатами этого оператора;

2) заключительный оператор - слово вида *стоп*$(\tau_1,..., \tau_n)$, где $n\geq 0$, а $\tau_1,..., \tau_n$ - термы; вхождения переменных в термы называют аргументами этого оператора;

3) оператор присваивания - слово вида $x:= \tau$, где x - переменная, а τ - терм; вхождения переменных в терм называют аргументами, а переменную x - результатом этого оператора;

4) условный оператор (тест) - логическое выражение; вхождения переменных в логическое выражение называют аргументами этого оператора;

5) оператор петли - односимвольное слово *петля*.

Среди операторов присваивания выделяют случаи: если τ - переменная, то оператор называют пересылкой ($x:=y$) и если τ -константа, то оператор называют засылкой ($x:=a$).

1.2.2 Графовая форма стандартной схемы

Определение 1.5. Графовой формой стандартной схемы в базисе ß называется конечный (размеченный ориентированный) граф без свободных дуг и с вершинами следующих пяти типов.

1 Начальная вершина, помеченная начальным оператором, из которой выходит ровно одна дуга. В схеме имеется ровно одна начальная вершина и нет дуг, ведущих к ней.

2 Заключительная вершина, помеченная заключительным оператором, из которой не выходит ни одной дуги. В схеме может быть несколько заключительных вершин.

3 Вершина-преобразователь, помеченная оператором присваивания, из которой выходит ровно одна дуга.

4 Вершина-распознаватель, помеченная условным оператором (называемым в этом случае также условием этой вершины), из которой выходят ровно две дуги, одна из них помечена символом 1 и называется 1-дугой, а другая помечена символом 0 и называется 0-дугой.

5 Вершина-петля, помеченная оператором петли, из которой не выходит ни одной дуги.

Вершины именуются (метки вершины) целым неотрицательным числом (0, 1, 2, ...). Начальная вершина всегда помечается меткой 0. Вершины изображаются прямоугольниками, а вершина-распознаватель - овалом. Операторы записываются внутри вершины.

Схема S будет правильной, если на каждой дуге заданы все переменные. Пример правильной ССП S_1 в графовой форме приведен на рисунке 1.1.

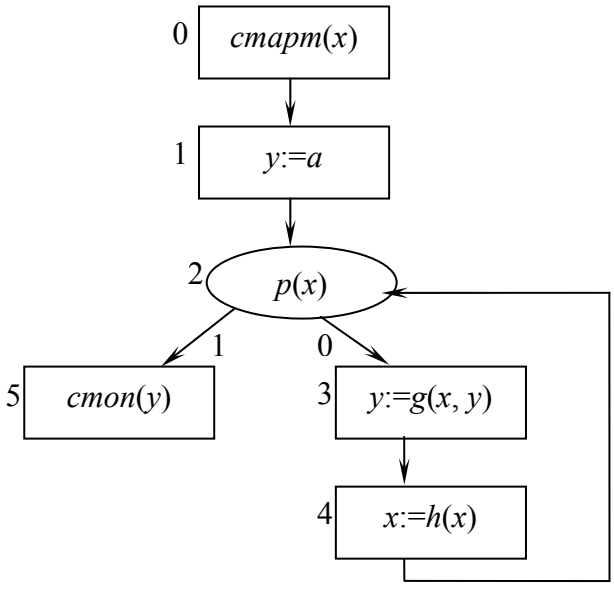


Рисунок 1.1 – Правильная стандартная схема программы S_1в графовой форме

1.2.3 Линейная форма стандартной схемы

Множество специальных символов расширим дополнительными новыми символами {:, *на*, *если*, *то*, *иначе*}.

Определение 1.6. Стандартная схема в линейной форме представляет собой последовательность инструкций, которая строится следующим образом:

1) если выходная дуга начальной вершины с оператором *старт*($x_1,\ldots,x_n$) ведет к вершине с меткой l, то начальной вершине соответствует инструкция:

0: *старт*($x_1,\ldots, x_n$) *на* l;

2) если вершина схемы S с меткой l - преобразователь с оператором присваивания $x:=\tau$, выходная дуга которого ведет к вершине с меткой l_1, то этому преобразователю соответствует инструкция:

l: $x:=\tau$ *на* l_1;

3) если вершина схемы S с меткой l - заключительная вершина с оператором *стоп*($\tau_1,\ldots, \tau_m$), то ей соответствует инструкция:

l: *стоп*($\tau_1,\ldots, \tau_m$);

4) если вершина схемы S с меткой l - распознаватель с условием $p(\tau_1,\ldots, \tau_m)$, причем 1-дуга ведет к вершине с меткой l_1, а 0-дуга - к вершине с меткой l_0, то этому распознавателю соответствует инструкция:

l: *если* $p(\tau_1,\ldots, \tau_m)$ *то* l_1 *иначе* l_0;

5) если вершина схемы S с меткой l - петля, то ей соответствует инструкция:

l: *петля*.

Часто используется сокращенная запись линейной формы за счет опускания меток и переходов на следующую инструкцию.

Пример 1.3. Полная и сокращенная линейные формы ССП, представленной на рисунке 1.1, будут иметь вид.

0: *старт*(x) *на* 1,	*старт*(x),
1: $y:=a$ *на* 2,	$y:=a$,
2: *если* $p(x)$ *то* 5 *иначе* 3,	2: *если* $p(x)$ *то* 5 *иначе* 3,
3: $y:=g(x, y)$ *на* 4,	3: $y:=g(x, y)$,
4: $x:=h(x)$ *на* 2,	$x:=h(x)$ *на* 2,
5: *стоп*(y).	5: *стоп*(y).

1.2.4 Интерпретация стандартных схем программ

Пусть задан некоторый базис ß, в котором определен класс стандартных схем.

Определение 1.7. Интерпретацией базиса ß в области интерпретации D называется функция I, которая сопоставляет:

1) каждой переменной x из базиса ß - некоторый элемент $d = I(x)$ из области интерпретации D;

2) каждой константе a из базиса - некоторый элемент $d = I(a)$ из D;

3) каждому функциональному символу $f^{(n)}(n \geq 1)$ - всюду определенную функцию $F^{(n)} = I(f^{(n)})$: $D^n \rightarrow D$;

4) каждой логической константе $p^{(0)}$ - один из символов множества {0, 1};

5) каждому предикатному символу $p^{(n)}$, где $n \geq 1$ - всюду определенный предикат $P^{(n)} = I(p^{(n)})$: $D^n \rightarrow \{0, 1\}$.

Определение 1.8. Пара (S, I), где S - схема в базисе ß, а I - интерпретация этого базиса, называется интерпретированной стандартной схемой, или стандартной программой.

Определим понятие выполнения программы.

Определение 1.9. Конечное множество переменных схемы S составляют ее память X_S.

Определение 1.10. Состоянием памяти программы (S, I) назовем функцию W: $X_s \rightarrow D$, которая каждой переменной x из памяти схемы S сопоставляет элемент $W(x)$ из области интерпретации D. Элемент $W(x)$ называется значением переменной x в состоянии W; запись $W \overset{x}{=} W'$ означает, что два состояния памяти W и W' совпадают с точностью до значений переменной x.

Начальным состоянием памяти X_s при интерпретации I назовем состояние W_0 такое, что $W_0(x) = I(x)$ для любой переменной x из X_s.

Значение терма τ при интерпретации I и состоянии памяти W (обозначим $\tau_I(W)$) определяется следующим образом:

1) если $\tau = x$, x – переменная, то $\tau_I(W) = W(x)$;

2) если $\tau = a$, a – константа, то $\tau_I(W) = I(a)$;

3) если $\tau = f^{(n)}(\tau_1, \tau_2, ..., \tau_n)$, то $\tau_I(W) = I(f^{(n)})(\tau_{1I}(W), \tau_{2I}(W), ..., \tau_{nI}(W))$.

Аналогично определяется значение теста π при интерпретации I и состоянии памяти W или $\pi_I(W)$:

если $\pi = p^{(n)}(\tau_1, \tau_2, ..., \tau_n)$, то $\pi_I(W) = I(p^{(n)})(\tau_{1I}(W), \tau_{2I}(W), ..., \tau_{nI}(W))$, $n \geq 0$.

Определение 1.11. Конфигурацией программы называют пару $U = (L, W)$, где L - метка вершины схемы S, а W - состояние ее памяти. Выполнение программы описывается конечной или бесконечной последовательностей конфигураций, которую называют протоколом выполнения программы (ПВП).

Определение 1.12. Протокол $(U_0, U_1, ..., U_i, U_{i+1}, ...)$ выполнения программы (S, I) определим следующим образом (ниже l_i означает метку вершины, а W_i - состояние памяти в i-й конфигурации протокола, $u_i = (l_i, W_i)$).

1 Начальная конфигурация $u_0 = (0, W_0)$, где W_0 - начальное состояние памяти схемы S при интерпретации I.

2 Пусть $u_i = (l_i, W_i)$ - i-я конфигурация протокола, а O - оператор схемы S в вершине с меткой l_i. Если O - заключительный оператор $cmon(\tau_1, ..., \tau_n)$, то u_i - последняя конфигурация протокола и протокол конечен. В этом случае говорят, что программа (S, I) останавливается, а последовательность значений $\tau_{1I}(W)$, $\tau_{2I}(W), \ldots, \tau_{nI}(W)$ объявляют результатом $val(S, I)$ выполнения программы (S, I). В противном случае, т.е. когда O - не заключительный оператор, в протоколе

имеется следующая (i+1)-я конфигурация $u_{i+1} = (l_{i+1}, W_{i+1})$, тогда:

а) если O - начальный оператор, а выходящая из него дуга ведет к вершине с меткой k, то $l_{i+1} = k$ и $W_{i+1} = W_i$;

б) если O - оператор присваивания $x := \tau$, а выходящая из него дуга ведет к вершине с меткой k, то $l_{i+1} = k$, $W_{i+1} \overset{x}{=} W_i$, причем $W_{i+1}(x) = \tau_I(W_i)$;

в) если O - условный оператор π и $\pi_I(W_i) = \Delta$, где $\Delta \in \{0, 1\}$, а выходящая из этого распознавателя Δ-дуга ведет к вершине с меткой k, то $l_{i+1} = k$ и $W_{i+1} = W_i$;

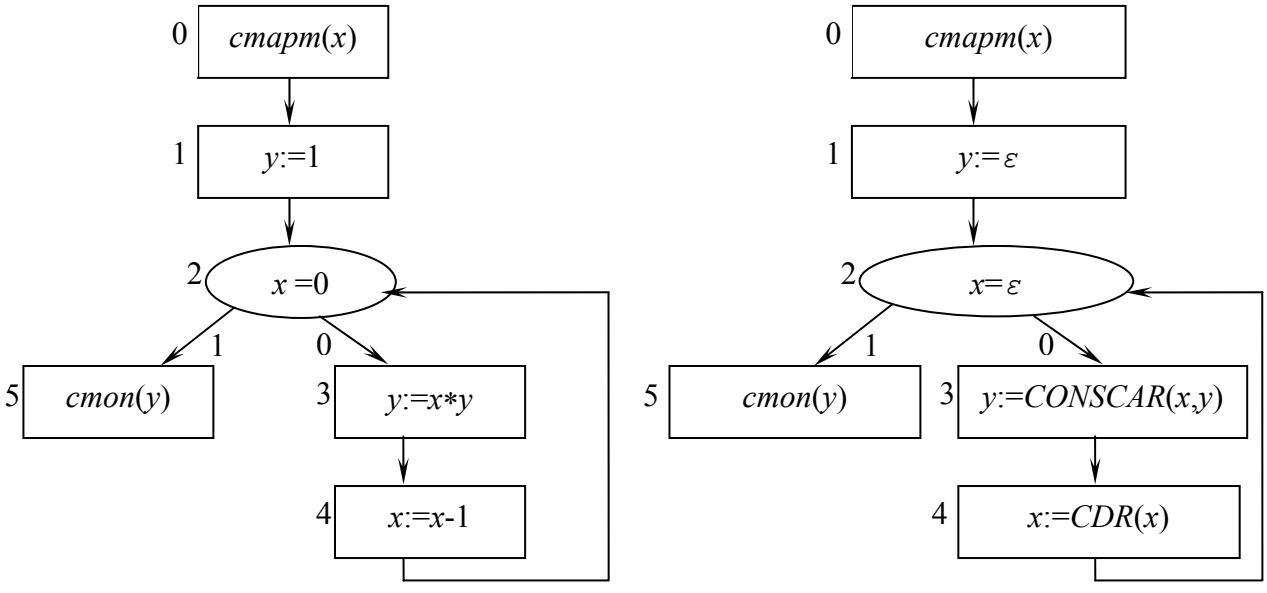
г) если O - оператор петли, то $l_{i+1} = l_i$ и $W_{i+1} = W_i$, так что протокол бесконечен.

Таким образом, программа останавливается тогда и только тогда, когда протокол ее выполнения конечен. В противном случае программа зацикливается и результат ее выполнения не определен.

Пример 1.4. Рассмотрим две интерпретации ССП S_1 (рисунок 1.1). Пусть интерпретация (S_1, I_1) задана следующим образом:

- область интерпретации D_1 - подмножество множества целых неотрицательных чисел;
- $I_1(x) = 4$; $I_1(y) = 0$; $I_1(a) = 1$;
- $I_1(g)=G$, где G - функция умножения чисел, т.е. $G(d_1, d_2)=d_1*d2$;
- $I_1(h)=H$, где H - функция вычитания единицы, т.е. $H(d)=d$-1;
- $I_1(p)=P_1$, где P_1 - предикат «равно 0», т.е. $P_1(d)=1$, если $d=0$.

Программа (S, I_1) вычисляет 4! (рисунок 1.2, а). Протокол выполнения программы (S_1, I_1) конечен (таблица 1.1), программа останавливается с результатом выполнения $val(S_1, I_1)=24$.



а) Программа (S_1, I_1) б) Программа (S_1, I_2)

Рисунок 1.2 – Интерпретации ССП S_1

Таблица 1.1 – Протокол выполнения программы (S_1, I_1)

Конфигурация		U_0	U_1	U_2	U_3	U_4	U_5	U_6	U_7	U_8	U_9	U_{10}	U_{11}	U_{12}	U_{13}	U_{14}	U_{15}
Метка		0	1	2	3	4	2	3	4	2	3	4	2	3	4	2	5
Значения	x	4	4	4	4	3	3	3	2	2	2	1	1	1	0	0	0
	y	0	1	1	4	4	4	12	12	12	24	24	24	24	24	24	24

Интерпретацию (S_1, I_2) зададим следующим образом:

- область интерпретации $D_2 = V^*$, где $V=\{k, l, m\}$, V^* - множество всех возможных слов в алфавите V, включая пустую строку ε;

- $I_2(x) = klm$, $I_2(y) = \varepsilon$, $I_2(a) = \varepsilon$;

- $I_2(g) = CONSCAR$ - суперпозиция функций *CONS* и *CAR* из языка Лисп, приписывающая первую букву первого слова ко второму слову, т.е. $CONSCAR(d\delta, \gamma) = d\gamma$, где $d \in V$, $\delta \in V^*$, $\gamma \in V^*$;

- $I_2(h) = CDR$ – функция, стирающая первую букву слова, т.е. $CDR(d\delta) = \delta$, где $d \in V$, $\delta \in V^*$;

- $I_2(p) = P_2$, где P_2 - предикат «равное ε», т.е. $P_2(\alpha)=1$, если $\alpha=\varepsilon$.

Программа (S_1, I_2) преобразует слово *klm* в слово *mlk* (рисунок 1.2, б). Протокол выполнения программы (S_1, I_2) конечен (таблица 1.2), программа останавливается с результатом выполнения $val(S_1, I_2)=mlk$.

Таблица 1.2 – Протокол выполнения программы (S_1, I_2)

Конфигурация		U_0	U_1	U_2	U_3	U_4	U_5	U_6	U_7	U_8	U_9	U_{10}	U_{11}	U_{12}
Метка		0	1	2	3	4	2	3	4	2	3	4	2	5
Значения	x	*klm*	*klm*	*klm*	*klm*	*lm*	*lm*	*lm*	*m*	*m*	*m*	ε	ε	ε
	y	ε	ε	ε	*k*	*k*	*k*	*lk*	*lk*	*lk*	*mlk*	*mlk*	*mlk*	*mlk*

1.3 Свойства и виды стандартных схем программ

1.3.1 Эквивалентность, тотальность, пустота, свобода

Определение 1.13. ССП S в базисе ß тотальна (пуста), если для любой интерпретации I базиса ß программа (S, I) останавливается (зацикливается).

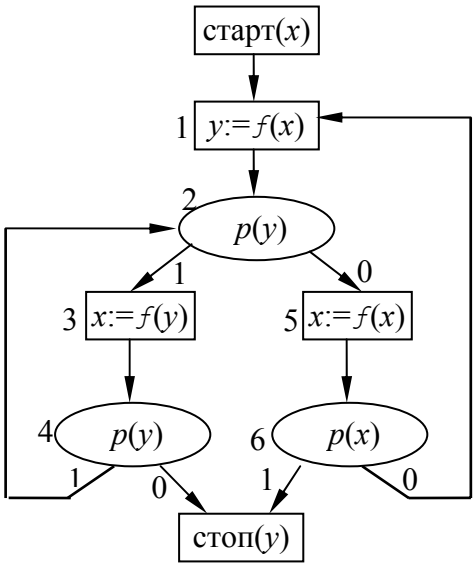
Определение 1.14. Стандартные схемы S_1 и S_2 в базисе ß функционально эквивалентны ($S_1 \sim S_2$), если либо обе зацикливаются, либо обе останавливаются с одинаковым результатом, т.е. $val(S_1, I) \approx val(S_2, I)$.

Пример 1.5. Тотальные, пустые и эквивалентные схемы S_2, S_3, S_4, S_5 приведены на рисунке 1.3.

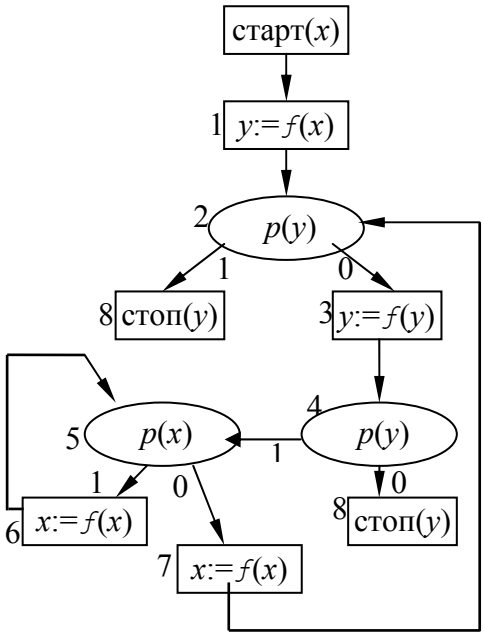
Определение 1.15. Цепочкой стандартной схемы называют:

1) конечный путь по вершинам схемы, ведущий от начальной вершины к заключительной;

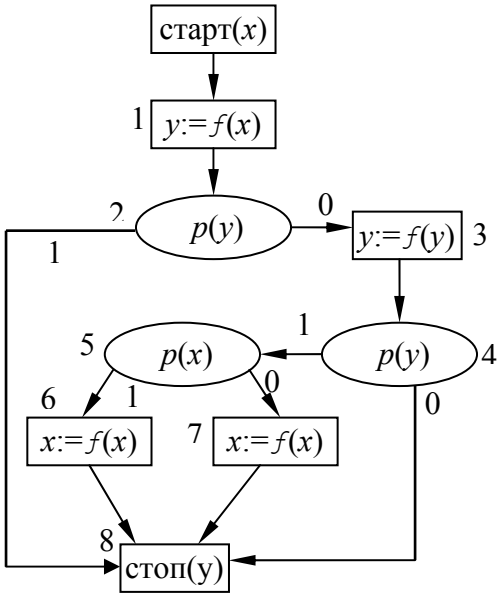
2) бесконечный путь по вершинам, начинающийся начальной вершиной схемы.



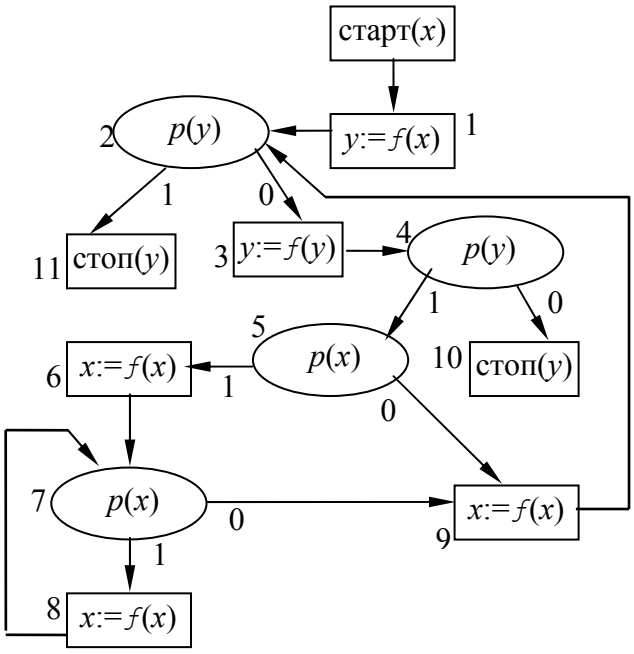
а) ССП S_2



б) СПП S_3



в) ССП S_4



г) ССП S_5

Рисунок 1.3 – Примеры тотальных, пустых и эквивалентных схем

Для вершины-распознавателя, то дополнительно указывается верхний индекс (1 или 0), определяющий 1-дугу или 0-дугу, исходящую из вершины.

Пример 1.6. Цепочками схемы S_1 (рисунок 1.1) являются (0, 1, 2^1, 5), (0, 1, 2^0, 3, 4, 2^0,3,4,2^1,5) и т. д.

Определение 1.16. Цепочкой операторов называется последовательность операторов, метящих вершины некоторой цепочки схемы.

Предикатные символы цепочки операторов обозначаются так же, как вершины распознавателей в цепочке схемы.

Пример 1.7. Для схемы S_1 цепочками операторов являются (*старт*(x), y:=a, $p^1(x)$, *стоп*(y)) или (*старт*(x), y:=a, $p^0(x)$, y:=$g(x, y)$, x:=$h(x)$, $p^0(x)$, y:=$g(x, y)$, x:=$h(x)$, $p^0(x)$, y:=$g(x, y)$, x:=$h(x)$, …)) и т. д.

Пусть S - ССП в базисе ß, I - некоторая его интерпретация, цепочка схемы (0, 1, l_2, l_3,…) – это последовательность меток инструкций S, выписанных в том порядке, в котором эти метки входят в конфигурации протокола выполнения программы (S, I). Говорят, что интерпретация I подтверждает (порождает) эту цепочку.

1.3.2 Свободные интерпретации

Множество всех интерпретаций велико и поэтому вводится класс свободных интерпретаций, который образует ядро класса всех интерпретаций в том смысле, что справедливость высказываний о семантических свойствах ССП достаточно продемонстрировать для программ, получаемых только с помощью свободных интерпретаций.

Все свободные интерпретации базиса ß имеют одну и ту же область интерпретации, которая совпадает со множеством T всех термов базиса ß. Все свободные интерпретации одинаково интерпретируют переменные и функциональные символы, а именно:

а) для любой переменной x из базиса ß и для любой свободной интерпретации I_h этого базиса $I_h(x)=x$;

б) для любой константы a из базиса ß $I_h(a)=a$;

в) для любого функционального символа $f^{(n)}$ из базиса ß, где $n\geq 1$, $I_h(f^{(n)})=F^{(n)}$: $T^n \rightarrow T$, где $F^{(n)}$ - словарная функция такая, что $F^{(n)}(\tau_1, \tau_2, ..., \tau_n) = f^{(n)}(\tau_1, \tau_2, ..., \tau_n)$, т. е. функция $F^{(n)}$ по термам $\tau_1, \tau_2, ..., \tau_n$ из T строит новый терм, используя функциональный смысл символа $f^{(n)}$.

г) интерпретация предикатных символов полностью свободна, т.е. разные свободные интерпретации различаются лишь интерпретацией предикатных символов.

После введения свободной интерпретации термы используются в двух разных качествах: как функциональные выражения в схемах и как значения переменных и выражений. В дальнейшем термы-значения будем заключать в апострофы. Например, если τ_1='$f(x, a)$' - терм-значение переменной x, а τ_2 ='$g(y)$' - терм-значение переменной y, то значение свободно интерпретированного терма $\tau_3=f(x, h(y))$ равно терму-значению '$f(f(x, a), h(g(y)))$'.

Пример 1.9. Пусть I_h – свободная интерпретация базиса, в котором определена схема S_1 (рисунок 1.1), и в этой интерпретации предикат $P=I_h(p)$ задан так: $P(\tau) = 1$, если число функциональных символов в τ больше двух; $P(\tau) = 0$, в противном случае. Тогда протокол выполнения программы (S_1, I_h) можно представить таблицей 1.3.

Таблица 1.3 – Протокол выполнения программы (S_1, I_h)

Конфигурация	Метка	Значения	
		x	y
U_0	0	$'x'$	$'y'$
U_1	1	$'x'$	$'a'$
U_2	2	$'x'$	$'a'$
U_3	3	$'x'$	$'g(x, a)'$
U_4	4	$'h(x)'$	$'g(x, a)'$
U_5	2	$'h(x)'$	$'g(x, a)'$
U_6	3	$'h(x)'$	$'g(h(x), g(x, a))'$
U_7	4	$'h(h(x))'$	$'g(h(x), g(x, a))'$
U_8	2	$'h(h(x))'$	$'g(h(x), g(x, a))'$
U_9	3	$'h(h(x))'$	$'g(h(h(x)), g(h(x), g(x, a)))'$
U_{10}	4	$'h(h(h(x)))'$	$'g(h(h(x)), g(h(x), g(x, a)))'$
U_{11}	2	$'h(h(h(x)))'$	$'g(h(h(x)), g(h(x), g(x, a)))'$
U_{12}	5	$'h(h(h(x)))'$	$'g(h(h(x)), g(h(x), g(x, a)))'$

Если из терма удалить все скобки и запятые, то получим слово (назовем его бесскобочным термом), по которому можно однозначно восстановить первоначальный вид терма (при условии, что отмечена или известна местность функциональных символов). Например, терму $g^{(2)}(h^{(1)}(x), g^{(2)}(x, y))$ соответствует бесскобочный терм $ghxgxy$.

1.3.3 Согласованные свободные интерпретации

Определение 1.17. Интерпретация I и свободная интерпретация I_h того же базиса ß согласованы, если для любого логического выражения π справедливо $I_h(\pi)=I(\pi)$.

Пример 1.10. Пусть $\tau = 'g(h(h(x)), g(h(x), g(x, a)))'$, а интерпретация I_3 совпадает с интерпретацией I_1 из примера 1.4 за исключением того, что $I(x)=3$. Так как $I_3(a) = 1$; $I_3(g)$ - функция умножения; $I_3(h)$ - функция вычитания 1; получаем $I_3(\tau) = ((3\text{-}1)\text{-}1)*((3\text{-}1)*(3*1)) = 6$.

В этом случае I_h примера 1.9 согласована с только что рассмотренной интерпретацией I_3, а так же с I_2 (рисунок 1.2, б), но не согласована с I_1 (рисунок 1.2, а).

Лемма 1.1. Справедливы прямое и обратное утверждения, что для каждой интерпретации I базиса ß существует согласованная с ней свободная интерпретация этого базиса.

Пример 1.11. Чтобы сделать I_h из примера 1.10 согласованной с I_1, необходимо условие $p(\tau)$ видоизменить: $p(\tau) = 1$, если число функциональных символов в τ больше трех; $p(\tau) = 0$, в противном случае.

Можно поступить и по другому. Оставить I_h без изменения и согласовать с ней I_1. Для этого необходимо будет заменить $I_1(x) = 4$ на $I_1(x) = 3$.

Введем вспомогательное понятие подстановки термов.

Определение 1.18. Если $x_1, \ldots, x_n$ ($n \geq 0$) – попарно различные переменные, а $\tau_1, \ldots, \tau_n$ – термы из T, π - функциональное или логическое выражение, то через $\pi[\tau_1/x_1, \ldots, \tau_n/x_n]$ будем обозначать выражение, получающееся из выражения π одновременной заменой каждого из вхождений переменной x_i на терм τ_i ($i = 1, \ldots, n$).

Пример 1.12. Справедливы следующие подстановки термов: $a[y/x]=a$, $f(x, y)[y/x, x/y]=f(y, x)$, $g(x)[g(x)/x]=g(g(x))$, $p(x)[a/x]=p(a)$.

В теории схем программ доказаны следующие основные теоремы о свободных интерпретациях.

Теорема 1.1. (Лакхэма – Парка – Паттерсона). Стандартные схемы S_1 и S_2 в базисе ß функционально эквивалентны тогда и только тогда, когда они функционально эквивалентны на множестве всех свободных интерпретаций базиса ß, т.е. когда для любой свободной интерпретации I_h программы (S_1, I_h) и (S_2, I_h) либо обе зацикливаются, либо обе останавливаются и $val(S_1,I_h)=val(S_2,I_h)$.

Теорема 1.2. Стандартная схема S в базисе ß пуста (тотальна, свободна) тогда и только тогда, когда она пуста (тотальна, свободна) на множестве всех свободных интерпретаций этого базиса, т.е. если для любой свободной интерпретации I_h программа (S, I_h) зацикливается (останавливается).

Теорема 1.3. Стандартная схема S в базисе ß свободна тогда и только тогда, когда она свободна на множестве всех свободных интерпретаций этого базиса, т.е. когда каждая цепочка схемы подтверждается хотя бы одной свободной интерпретацией.

1.3.4 Логико-термальная эквивалентность

Определение 1.19. Отношение эквивалентности E, заданное на парах стандартных схем, называют **корректным**, если для любой пары схем S_1 и S_2 из $S_1 \overset{E}{\sim} S_2$ следует, что $S_1 \sim S_2$, т. е. S_1 и S_2 функционально эквивалентны.

Поиск разрешимых корректных отношений эквивалентности представляет значительный интерес с точки зрения практической оптимизации преобразования программ, поскольку в общем виде функциональная эквивалентность стандартных схем алгоритмически неразрешима.

Идея построения таких (корректных и разрешимых) отношений связанна с введением понятия истории цепочки схем. В истории с той или иной степенью детальности фиксируются промежуточные результаты выполнения операторов рассматриваемой цепочки. Эквивалентными объявляются схемы, у которых совпадают множества историй всех конечных цепочек.

Одним из отношений эквивалентности является логико-термальная эквивалентность.

Определим термальное значение переменной x для конечного пути ω схемы S как терм $t(\omega, x)$, который строится следующим образом.

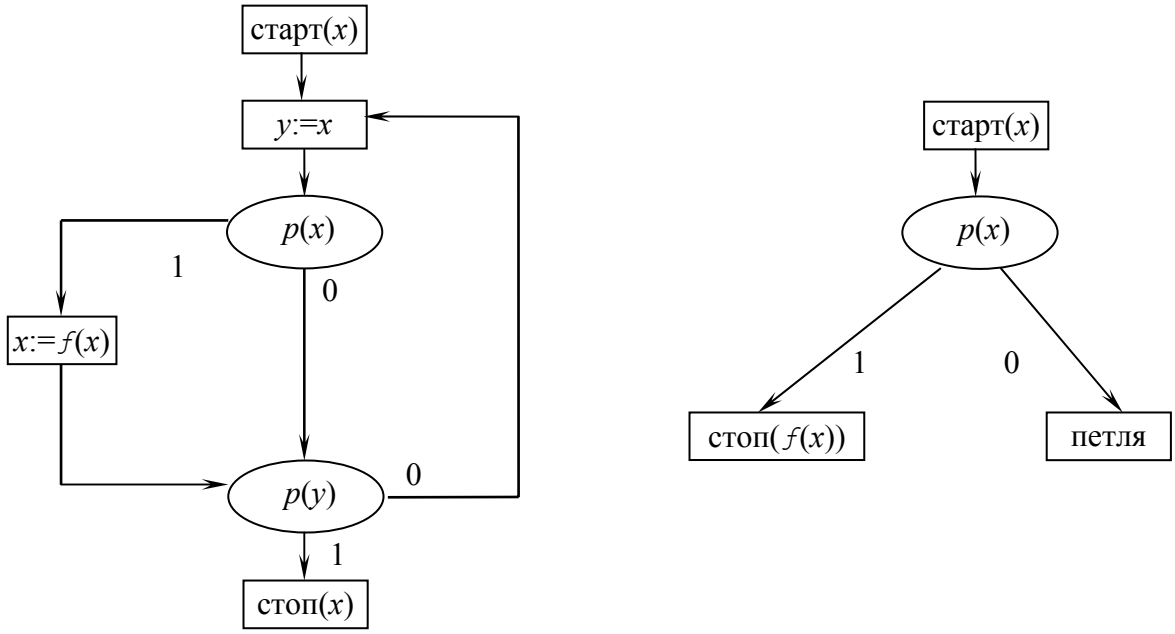
1 Если путь ω содержит только один оператор A, то $t(\omega, x) = \tau$, если A – оператор присваивания и $t(\omega, x) = x$, в остальных случаях.

2 Если $\omega = \omega' Ae$, где A – оператор, e – выходящая из него дуга, ω' – непустой путь, ведущий к A, а $x_1, \ldots, x_n$ – все переменные терма $t(Ae, x)$, то $t(\omega, x) = t(Ae, x)[t(\omega', x_1)/x_1, \ldots, t(\omega', x_n)/x_n]$.

Понятие термального значения распространим на произвольные термы τ.

$$t(\omega, x) = \tau\,[t(\omega, x_1)/x_1, \ldots, t(\omega, x_n)/x_n].$$

Пример 1.13. Пути *старт*(x); $y{:=}x$; $p^1(x)$; $x{:=}f(x)$; $p^0(y)$; $y{:=}x$; $p^1(x)$; $x{:=}f(x)$ в схеме на рисунке 1.4 а) соответствует термальное значение $f(f(x))$ переменной x.



а) ССП S_6 б) ССП S_7

Рисунок 1.4 – Функционально эквивалентные схемы

Определение 1.20. Для пути ω в стандартной схеме S его логико-термальная история $lt(S, \omega)$ определяется как слово, которое строится следующим образом.

1 Если путь ω не содержит распознавателей и заключительной вершины, то $lt(S, \omega)$ – пустое слово.

2 Если $\omega = \omega' ve$, где v – распознаватель с тестом $p(\tau_1, \ldots, \tau_k)$, а e – выходящая из него Δ-дуга, $\Delta \in \{0, 1\}$, то $lt(S, \omega) = lt(S, \omega')\, p^{\Delta}(t(\omega', \tau_1), \ldots, t(\omega', \tau_k))$.

3 Если $\omega = \omega' v$, где v – заключительная вершина с оператором *стоп*$(\tau_1, \ldots, \tau_k)$, то $lt(S, \omega) = lt(S, \omega')\, t(\omega', \tau_1) \ldots t(\omega', \tau_k)$.

Пример 1.14. Логико-термальной историей пути, приведенного в примере 1.13, будет слово $p^1(x)\, p^0(x)\, p^1(f(x))$.

Определение 1.21. Детерминантом (обозначение: *det*(*S*)) стандартной схемы *S* называют множество логико-термальных историй всех ее цепочек, завершающихся заключительным оператором.

Схемы S_1 и S_2 называют логико-термально эквивалентными (лт - эквивалентными, обозначение $S_1 \overset{ЛТ}{\sim} S_2$), если их детерминанты совпадают.

В теории схем программ доказано, что лт- эквивалентность корректна, т.е. из лт- эквивалентности следует функциональная эквивалентность ($S_1 \overset{ЛТ}{\sim} S_2 \Rightarrow S_1 \sim S_2$). Обратное утверждение не верно. Действительно, на рисунке 1.4 приведены две функционально эквивалентные схемы, которые не являются лт-эквивалентными.

ЛТ–эквивалентность допускает меньше сохраняющих ее преобразований, чем функциональная эквивалентность.

1.4. Моделирование стандартных схем программ

1.4.1 Одноленточный автомат

Конечный одноленточный (детерминированный, односторонний) автомат обнаруживает ряд полезных качеств, используемых в теории схем программ для установления разрешимости свойств ССП.

Определение 1.22. Одноленточным конечным автоматом называется пятерка объектов:

$$M = (Q, T, F, H, Z),$$

где Q - конечное множество состояний автомата;

T - конечное множество допустимых входных символов;

F - функция переходов, задающая отображение $Q \times T \rightarrow Q$, т.е. $F(q,t) = p$, где $q, p \in Q, t \in T$;

H - конечное множество начальных состояний автомата;

Z - множество заключительных состояний автомата, $Z \subseteq Q$.

В контексте моделирования стандартных схем программ рассматриваются проблемы пустоты и эквивалентности автоматов.

Определение 1.23. Автомат называется пустым, если множество допускаемых им строк L_M пусто.

Определение 1.24. Автоматы M_1 и M_2 эквивалентны, если совпадают множества допускаемых ими строк, т.е. $L_{M1} = L_{M2}$.

Для одноленточного конечного автомата доказаны следующие теоремы.

Теорема 1.4. Проблема пустоты одноленточного конечного автомата разрешима.

Доказательство основано на проверке допустимости конечного множества всех слов, длина которых не превышает числа состояний ОКА - *n*. Если ни одно слово из этого множества не допускается, то ОКА «пуст».

Теорема 1.5. Проблема эквивалентности ОКА разрешима.

Доказательство основано на использовании отношения эквивалентности двух состояний q и q': если состояния q и q' эквивалентны, то для всех $t \in T$ состояния $F(q, t)$ и $F'(q', t)$ также эквивалентны. В формируемые пары не должны входить одновременно заключительное и незаключительное состояния.

1.4.2 Многоленточный автомат

Многоленточный (детерминированный, односторонний) конечный автомат (МКА) задается по аналогии с ОКА. Отличие состоит в том, что множество состояний Q разбивается на $n \geq 2$ непересекающихся подмножеств Q_1, ..., Q_n. «Физическая» интерпретация n - ленточного автомата отличается тем, что он имеет n лент и n головок, по головке на ленту. Если автомат находится в состоянии $q \in Q_i$, то i-я головка читает i-ю ленту так же, как это делает ОКА. При переходе МКА в состояние $q' \in Q_j$ $(i \neq j)$ i-я головка останавливается, а j-я начинает читать свою ленту. МКА останавливается, когда головка на одной из лент прочитает символ #.

Пример 1.15. Рассмотрим функционирование МКА с n=2 (рисунок 1.5) на примере сравнения пар слов в алфавите {1, 0} и допуске только совпадающих слов.

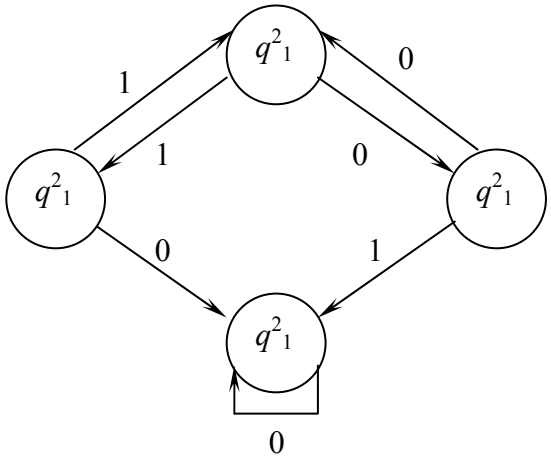


Рисунок 1.5 – Граф двухленточного конечного автомата

Здесь $Q=Q_1 \cup Q_2$, где $Q_1=\{q_0^1\}$; $Q_2=\{q_1^2, q_2^2, q_3^2\}$; $Z=\{q_0^1\}$; $T=\{0, 1\}$, начальное состояние $H=\{q_0^1\}$. МКА обрабатывает наборы слов (U_1, U_2), где слово U_1 записано на первой ленте, а U_2 - на второй. Допустимое множество строк L_M - это все возможные пары одинаковых слов, т.е. наборы, где $U_1 = U_2$. Например, набор может быть (1101, 1101) и т.п.

В том случае, когда слова совпадают, МКА остановится в заключительном состоянии q_0^1 (именно в этом состоянии поступит символ #) и набор будет допущен. Если слова не совпадут хотя бы в одном символе, МКА перейдет в состояние q_3^2 и не выйдет из него до появления символа #, который и зафиксирует отсутствие совпадения слов, т.е. не допустит искаженный набор.

Проблема эквивалентности и пустоты двухленточных автоматов разрешима.

1.4.3 Двухголовочный автомат

Двухголовочный конечный автомат (ДГКА) имеет одну ленту и две головки, которые могут независимо перемещаться вдоль ленты в одном направлении. Множество состояний Q разбито на два непересекающихся множества. В состояниях Q_1 активна первая головка, а в состояниях Q_2 - вторая.

Двухголовочный автомат можно рассматривать как такой двухленточный автомат, который работает с идентичными словами на обеих лентах.

Пример 1.16. Рассмотрим ДГКА, который проверяет равенство двух последовательно записанных слов в алфавите $V = \{0, 1\}$. Признаком окончания каждого из слов сделаем вспомогательный символ *, не входящий в V. Автомат должен допускать только слова вида $a^* a^*$, где $a \in V^*$. Возьмем автомат $M = (Q_1 \cup Q_2, V \cup \{*\}, F, q_0^1, q_7^1)$, где $Q_1 = \{q_0^1, q_1^1, q_2^1, q_7^1\}$ - множество состояний, в которых активна первая головка; $Q_2 = \{q_3^2, q_4^2, q_5^2, q_6^2\}$ - множество состояний, в которых активна вторая головка. Отношение F задано графом автомата на рисунке 1.6.

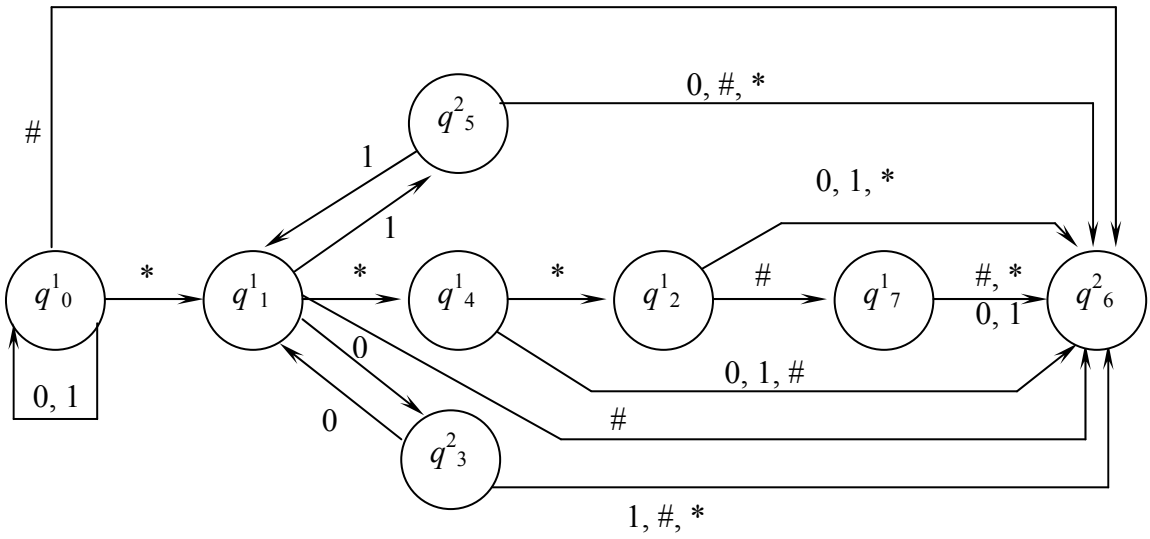


Рисунок 1.7 – Пример графа двухголовочного конечного автомата

Находясь в состоянии q_0^1, автомат передвигает первую головку к началу второго слова и, обнаружив его, переходит в состояние q_1^1. Если конец ленты # встречается раньше символа *, автомат переходит в незаключительное состояние q_6^2. Если же автомат приходит к состоянию q_1^1, он считывает поочередно символы второго слова первой головкой (состояние q_1^1), а символы первого слова — второй головкой (состояния q_3^2 и q_5^2), сравнивая эти символы. Автомат возвращается каждый раз в состояние q_1^1, если символы одинаковы. Если же обнаружится несовпадение символов или первая головка встречает конец слова раньше символа *, автомат уходит в состояние q_6^2. Попав в это состояние, автомат не может выйти из него; перемещая вторую головку к концу слова на

ленте, он достигает #, находясь в незаключительном состоянии, так что слово на ленте отвергается. Если первая головка достигнет конца второго слова, а вторая головка обнаружит, что первое слово тоже просмотрено до конца, то автомат перейдет в заключительное состояние q_7^1. В противном случае автомат перейдет в состояние q_6^2, отвергая слово.

Этот пример легко обобщить на случай произвольного алфавита V, увеличивая количество состояний множества Q.

Для двухголовочного конечного автомата доказаны следующие теоремы.

Теорема 1.6. Проблема пустоты ДГКА не является частично разрешимой.

Теорема 1.7. Проблема эквивалентности ДГКА не является частично разрешимой.

1.4.4 Двоичный двухголовочный автомат

Стандартные схемы могут моделировать (в уточненном ниже смысле) двухголовочные автоматы, что позволяет свести проблему пустоты этих автоматов к проблеме пустоты схем. Такое моделирование можно осуществить более простым способом, если использовать специальный класс двухголовочных автоматов, а именно класс двоичных автоматов, работающих со словами над алфавитом {0, 1}. Следующая лемма устанавливает связь между классом всех двухголовочных автоматов и подклассом двоичных автоматов специального вида.

Лемма 1.2. Существует алгоритм преобразования двухголовочных автоматов в двоичные двухголовочные автоматы (ДДГКА), сохраняющий пустоту автоматов (построенный двоичный автомат M_b пуст тогда и только тогда, когда пуст исходный автомат M).

Доказательство. Пусть ДГКА M над алфавитом $T = \{a_1, a_2, ..., a_n\}$ имеет множество состояний $Q_A=\{q_1^k, q_2^k, \ldots, q_n^k\}$, где верхний индекс ($k$ = 1, 2) определяет номер активной головки. Преобразование этого автомата в двоичный начнем с кодировки символов и слов из T^* словами в алфавите {0, 1} по следующему правилу:

- код(#) = 0;
- код(a_i) = $\underbrace{11\ldots1}_{i}0$, $a_i \in T$, (i = 1, …, n);
- код(αa_i) = код(α) код(a_i), $\alpha \in T^*$, $a_i \in T$.

Так как символ # кодируется нулем, то любому непустому слову на ленте автомата M соответствует двоичное слово на ленте автомата M_b, оканчивающееся двумя нулями. Автомат останавливается, прочитав два нуля подряд (или 0, означающий пустое слово).

Автомат M преобразуется в двоичный автомат M_b так, как показано на графах рисунка 1.7. Каждый фрагмент графа M (рисунок 1.7, а) заменяется фрагментом (рисунок 1.7, б) для M_b. После замены к графу добавляются фрагменты в и г рисунка 1.7.

Таким образом, множество состояний автомата M_b включает:

а) все старые состояния из Q_M;

б) для каждого старого состояния q_j^k n новых состояний, n - число символов алфавита T;

в) два новых состояния r_1^1 и r_2^1.

Заключительными состояниями автомата M_b являются заключительные состояния автомата M. Вершины S_a (останов допускающая) и S_r (останов отвергающая) носят на графе M_b вспомогательный характер. Они отмечают тот факт, что автомат прочитал два нуля подряд и остановился в заключительном состоянии (случай S_a) или в незаключительном состоянии (случай S_r).

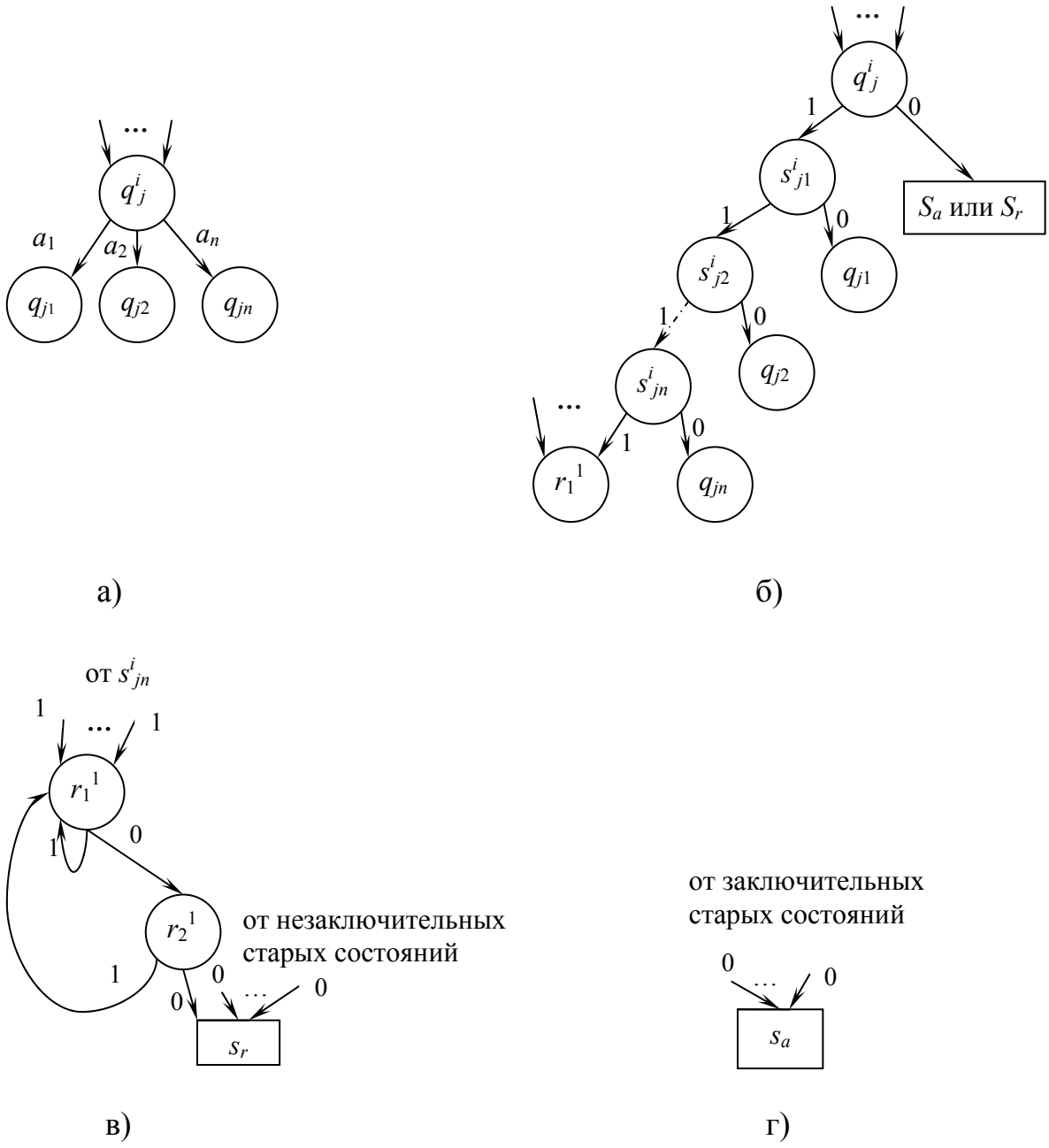


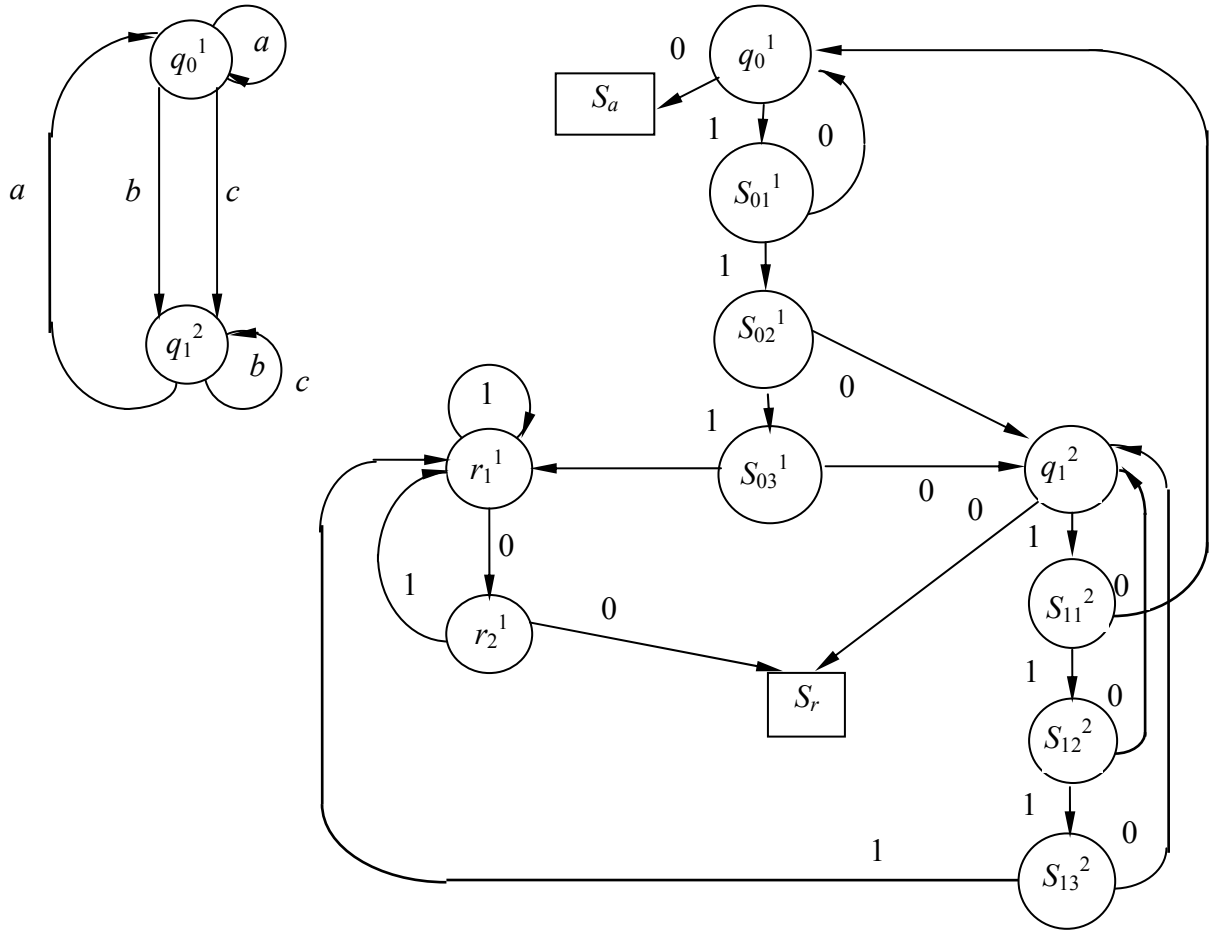
Рисунок 1.7 – Построение ДДГКА по ДГКА

Очевидно, что автомат M_b допускает двоичное слово p тогда и только тогда, когда оно является двоичным кодом слова из T^*, допускаемого автоматом

M. Таким образом, из пустоты автомата *M* следует пустота автомата M_b, и наоборот.

Пример 1.17. На рисунке 1.8 а приведен граф ДГКА *M*, допускающий только те слова в алфавите $T = \{a, b, c\}$, в которых символ *a* встречается не меньшее число раз, чем символы *b* и *c*, вместе взятые. Заключительное состояние $Z = \{q_0^{\,1}\}$. На рисунке 1.8 б приведен граф ДДГКА, построенный по автомату *M* (10 – код символа *a*, 110 – код *b*, 1110 – код *c*).

По заданному ДДГКА возможно построить ССП и наоборот, что позволяет решить задачу разрешимости (не разрешимости) свойств ССП, так как эта задача для ДДГКА решена.



а) ДГКА б) ДДГКА

Рисунок 1.8 – Граф исходного и результирующего автомата

1.4.5 Моделирование двоичного автомата стандартной схемой

Двоичное слово $\alpha = b_1 b_2 \ldots b_n$ согласовано со свободной интерпретацией базиса ß, если для любого *i*, $1 \le i \le n$, $I(p)('f^i \alpha') = b_i$, где *p* - единственный предикатный символ базиса ß.

Пример 1.18. Слово 101010100 согласовано с любой свободной интерпретацией I такой, что для всех $i \le 9$ $I(p)('f^i\alpha') = 1$ если i нечетно и меньше 9, и $I(p)('f^i\alpha') = 0$, если i четно или равно 9. Свободная интерпретация I такая, что для всех i $I(p)('f^i\alpha') = 0$, согласована с любым словом, не содержащим 1.

Для того чтобы свести проблему пустоты ДДГКА к проблеме пустоты стандартных схем покажем, что для любого двоичного автомата M можно построить схему S, которая моделирует автомат M в следующем смысле. Если на ленту автомата M подано произвольное двоичное слово α, то программа (S, I), где I - любая свободная интерпретация базиса ß, согласованная с α, останавливается в том и только в том случае, когда автомат допускает слово α.

В теории схем программ доказаны следующие утверждения.

Лемма 1.3. ДДГКА пуст в том и только в том случае, если пуста моделирующая его стандартная схема.

Лемма 1.4. Для любого ДДГКА можно построить моделирующую его стандартную схему.

Доказательство. Построение схемы осуществляется в два этапа. На первом этапе заготавливаются фрагменты стандартной схемы, из которых на втором этапе собирается схема. И автомат, и схема считаются представленными в графовой форме. Фрагмент графа - это его подграф (подмножество вершин вместе с дугами, соединяющими эти вершины). В рассматриваемых ниже фрагментах выделена начальная вершина (в нашем случае - это вершина, в которую не заходит ни одна дуга, или, в случае фрагмента с одной вершиной, - сама эта вершина). Фрагмент может иметь выходные дуги, не заходящие в его вершины. У стрелок выходных дуг есть специальные пометки: в случае фрагментов графа автомата - символы состояний, в случае фрагментов схемы - номера фрагментов. Пометки у выходных дуг нужны для сборки схемы из фрагментов на втором этапе преобразования.

Пусть граф двоичного двухголовочного автомата A включает вершины-состояния $\{q_1^{i_1}, q_2^{i_2}, ..., q_n^{i_n}\}$ и две вспомогательные вершины - останов допускающая и останов отвергающая - q_{n+1} и q_{n+2} соответственно.

Схема собирается из n+4 фрагментов $f_{\text{нач}}, f_0, f_1, \ldots, f_j, \ldots, f_n, f_{n+1}, f_{n+2}$. Фрагмент $f_{\text{нач}}$ – общий для всех схем начальный фрагмент, представленный на рисунке 1.9 а. Фрагменты f_j, $0 \le j \le n$ соответствуют состояниям $q_1^{i_1}$ и строятся по фрагментам графа автомата, связанным с $q_1^{i_1}$ так, как показано на рисунке 1.9 б (обратите внимание на соответствие пометок выходных дуг фрагмента автомата и фрагмента схемы). Фрагмент f_{n+1} соответствует вершине q_{n+1} (рисунок 1.9 в), фрагмент f_{n+2} - вершине q_{n+2} (рисунок 1.3 г). При любой интерпретации выполнение фрагмента на рисунке 1.9 г сводится к бесконечному выполнению распознавателя, независимо от логического выражения внутри него. Поэтому этот фрагмент эквивалентен оператору петля (рисунок 1.9 д).

На втором этапе из фрагментов $f_{\text{нач}}, f_0, f_1, \ldots, f_j, \ldots, f_n, f_{n+1}, f_{n+2}$ собирается стандартная схема по следующему правилу: выходная дуга с пометкой j фрагментов $f_{\text{нач}}, f_0, \ldots, f_n$ заводится на начальную вершину фрагмен-

та f_j, где $0 \le j \le n+2$. Остается перенумеровать подходящим образом все вершины полученного графа числами-метками, и стандартная схема построена.

Работе автомата A над некоторым двоичным словом α соответствует выполнение программы (S, I), где S - построенная схема, I - некоторая свободная интерпретация базиса ß, согласованная с α. Интуитивно это соответствие состоит в следующем. Команда $q^i_j b \to q_k$, где b - символ 0 или 1, заставляет автомат выполнить очередной шаг, состоящий из двух действий - перехода в новое состояние в зависимости от прочтенного символа b и сдвига головки. Шагом выполнения программы будем считать выполнение фрагмента.

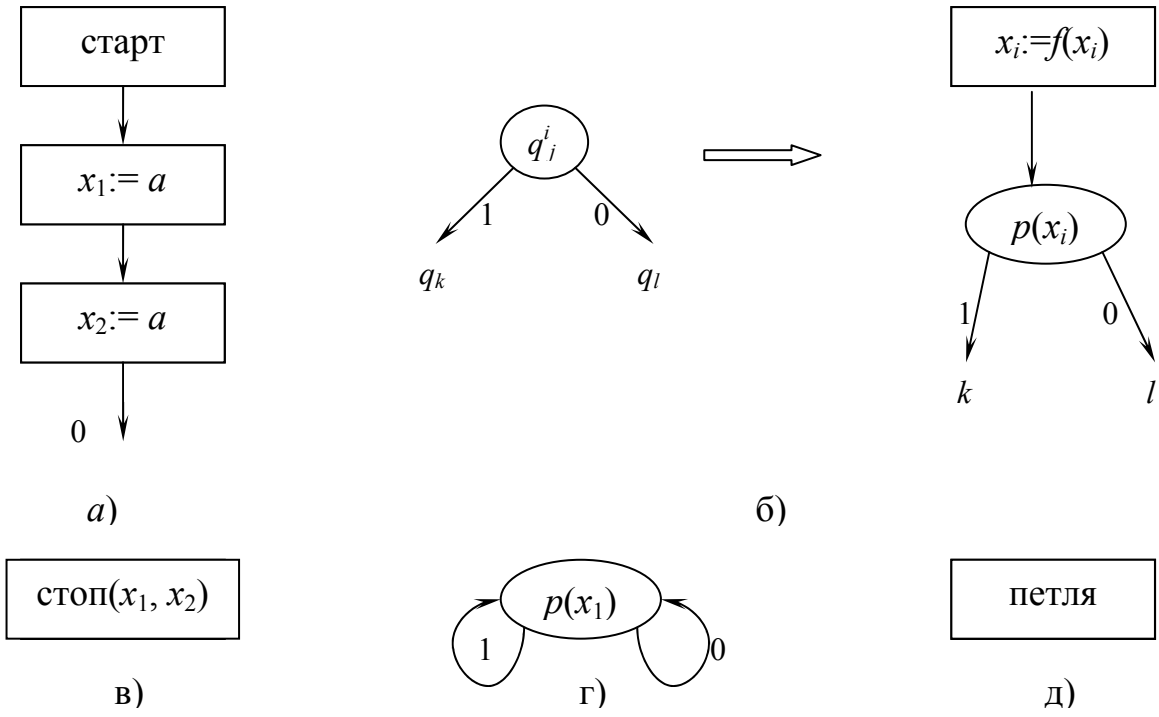


Рисунок 1.9 - Фрагменты стандартной схемы

Первый шаг - выполнение начального фрагмента $f_{нач}$, в результате чего значения обеих переменных x_1 и x_2 становятся равными 'α'. Каждый шаг выполнения фрагментов f_0, ..., f_n соответствует шагу работы автомата и также состоит из двух действий: сдвигу головки i соответствует выполнение оператора присваивания $x_i := I(f)(x_i)$, а переходу в новое состояние - выполнение условного оператора $I(p)(x_i)$. Если автомат остановился в заключительном состоянии, программа должна выполнить оператор стоп(x_1, x_2) и остановиться. Если автомат остановился в не заключительном состоянии, программа должна зациклиться, бесконечно выполняя фрагмент f_{n+2}.

Покажем, что построенная схема S моделирует автомат A. Для этого установим индукцией по числу шагов работы автомата над словом α, что после любого шага k имеет место следующее соответствие: если автомат A

после k -го шага перешел в состояние q^i_j, $0 \le j \le n+2$, а головки 1 и 2 прочитали к этому времени префиксы $b_1, b_2, \ldots, b_l$ и соответственно $b_1, b_2, \ldots, b_m$ слова α, то программа (S, I), где I - согласованная с α свободная интерпретация, после $(k + 1)$-го шага выполнила фрагмент f_j, а $W_{k+1}(x_1) = 'f^l\alpha'$ и $W_{k+1}(x_2) = 'f^m\alpha'$, где $W_{\kappa+1}$ - состояние памяти программы после $(k + 1)$-го шага.

Базис индукции. В начальный момент (после «нулевого шага») автомат находится в состоянии q^i_0 обе головки не прочитали еще ни одного символа. Программа сделала один шаг (выполнен начальный фрагмент $f_{\text{нач}}$) и переходит к выполнению фрагмента f_0, $W_1(x_1) = W_1(x_2) = '\alpha'$.

Шаг индукции. Пусть после k-го шага автомата имеет место предположенное соответствие между результатом работы автомата и результатом $(k+1)$-го шага выполнения программы. Покажем, что такое соответствие будет иметь место и после $(k+1)$-го шага автомата ($(k+2)$-го шага программы).

1 Пусть на $(k+1)$-м шаге автомат прочитал головкой i очередной символ b на ленте и перешел из состояния q^i_j в новое состояние q^{ir}_r. Для определенности положим, что $i = 1$. По построению схемы значение переменной x_1, после $(k+2)$-го шага программы станет равным $'f^{l+1}\alpha'$, а значение переменной x_2 не изменится. Терм $'f^{l+1}\alpha'$ - значение аргумента предиката $I(p)$ во фрагменте f_j. Так как слово α и свободная интерпретация I согласованы, то $I(p)('f^{l+1}\alpha') = b$. Поэтому на следующем шаге работы программы будет выполняться фрагмент f_r.

2 Если после k-го шага автомат остановился и q_j - заключительное состояние, то, по построению схемы, программа перейдет на $(k+2)$-м шаге к выполнению фрагмента f_{n+1}, т.е. выполнит оператор стоп(x_1, x_2) и остановится с результатом $val(S, I) = ('f^l\alpha', 'f^m\alpha')$, где l или m равно длине слова α.

3 Если после k-го шага автомат остановился и q_j – не заключительное состояние, то программа перейдет к бесконечному выполнению циклического фрагмента f_{n+2}.

Таким образом, для любого слова α на ленте автомата и для любой свободной интерпретации I, согласованной с α, программа (S, I) останавливается, если автомат допускает слово α, и зацикливается, если автомат отвергает a. Заметим, что если программа (S, I) останавливается, то результат $val(S, I) = ('f^l\alpha', 'f^m\alpha')$, где $max(l, m)$ равен длине слова α.

Пример 1.18. На рисунке 1.10 приведена стандартная схема, моделирующая двоичный двухголовочный автомат, изображенный на рисунке 1.8. Возле вершин-распознавателей выписаны соответствующие состояния автомата.

Доказаны следующие теоремы о неразрешимых свойствах стандартных схем программ.

Теорема 1.8 (Лакхэм - Парк - Патерсон). Проблема пустоты стандартных схем не является частично разрешимой.

Теорема 1.9 (Лакхэм - Парк - Патерсон, Летичевекай). Проблема функциональной эквивалентности стандартных схем не является частично разрешимой.

Теорема 1.10 (Лакхэм - Парк - Патерсон). Проблема тотальности стандартных схем частично разрешима.

Теорема 1.11 (Патерсон). Проблема свободы стандартных схем не является частично разрешимой.

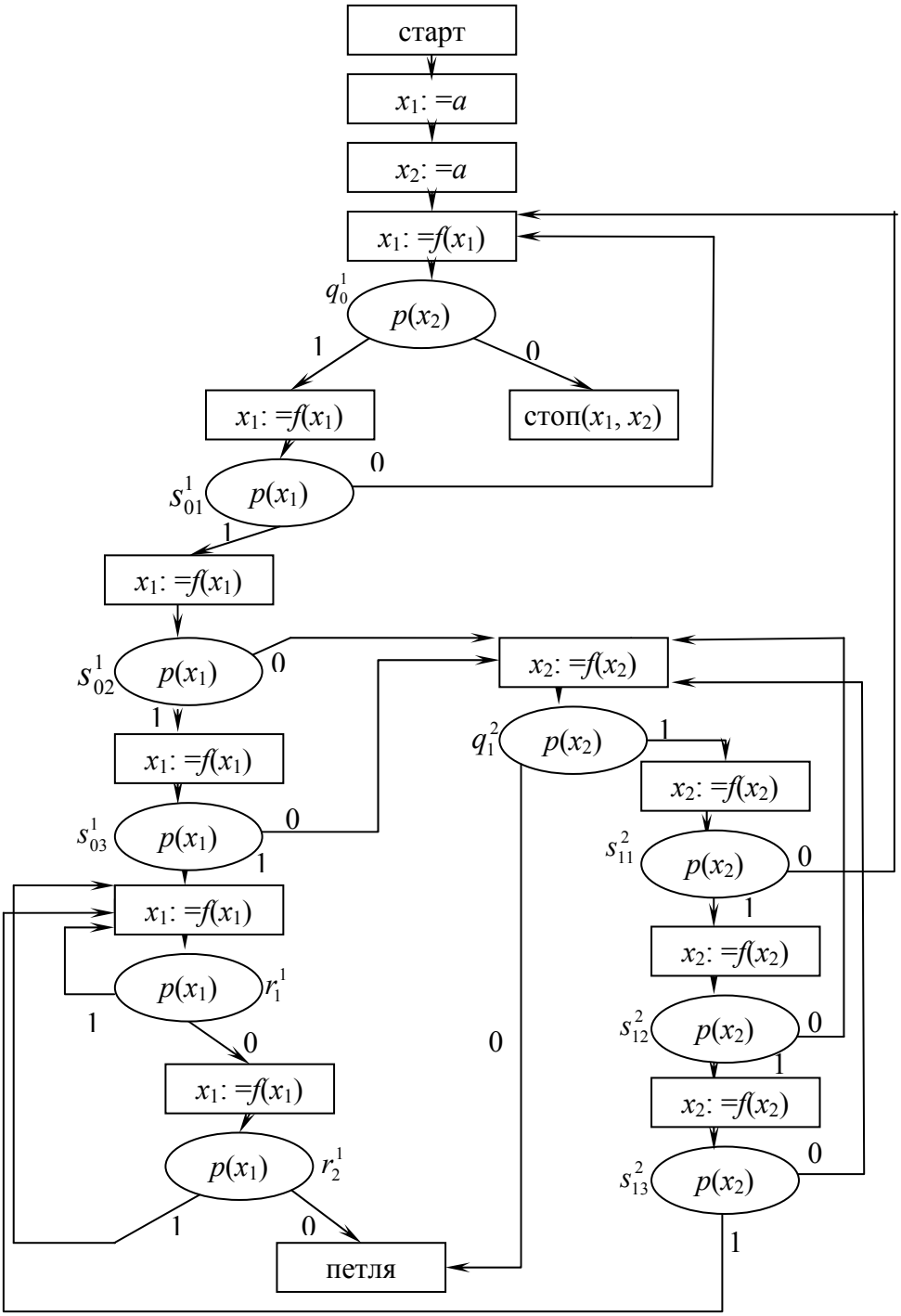


Рисунок 1.10 - Схема, моделирующая автомат

Вопросы и задания для самоконтроля

1 Сформулируйте требования, которым отвечают схемы программ.

2 Назовите главное отличие схем от программ.

3 Перечислите элементы полного базиса и множества операторов класса стандартных схем программ.

4 Приведите примеры 3-х местного терма и 4-х местного теста.

5 Перечислите типы вершин графовой формы стандартной схемы и соответствующие им инструкции линейной формы.

6 Дайте определение понятия интерпретация стандартной схемы программы.

7 Запишите правила построения протокола выполнения программы.

8 Для схемы программы S_8, представленной на рисунке 1.11, задайте конечную интерпретацию и постройте протокол выполнения полученной программы.

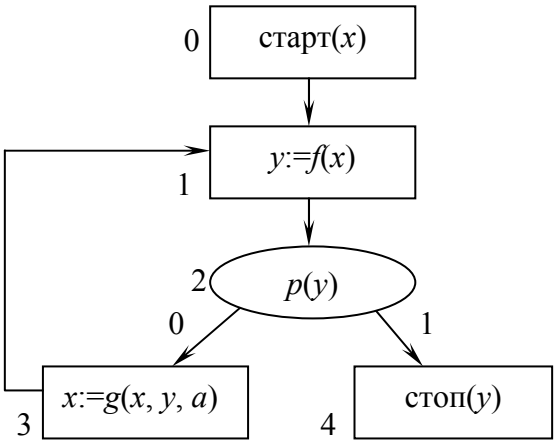


Рисунок 1.11 – Схемы программы S_8

9 Проверьте схему S_8 на тотальность и пустоту и постройте для нее функционально эквивалентную схему.

10 Для схемы S_8 постройте цепочку и цепочку операторов.

11 Каким образом трактуют элементы базиса его свободные интерпретации?

12 Вычислите значение свободно интерпретированного терма $\tau_3=g(a, h(x, y))$, если терм-значение x есть '$f(f(x))$', а терм-значение y есть '$g(x, a)$'.

13 Задайте для схемы S_8 свободную интерпретацию и постройте протокол выполнения полученной программы.

14 Выполните подстановку для терма $f(x, y, z)[h(y)/x, a/y, g(h(x, z))/z]$.

15 По схеме S_8 определите термальное значение переменной y для пути, соответствующего цепочке схемы 0, 1, 2^0, 3, 1, 2^1, 4 и логико-термальную историю этого пути.

16 Сравните «физические» интерпретации ОКА, МКА, ДГКА и ДДГКА.

17 Изложите алгоритм преобразования ДГКА в ДДГКА.

18 Назовите этапы построения стандартной схемы, моделирующей заданный ДДГКА.

19 Для ДГКА, заданного на рисунке 1.12, построить ДДГКА и стандартную схему, моделирующую его. Начальное и заключительное состояние автомата $H=Z=\{q_0^1\}$.

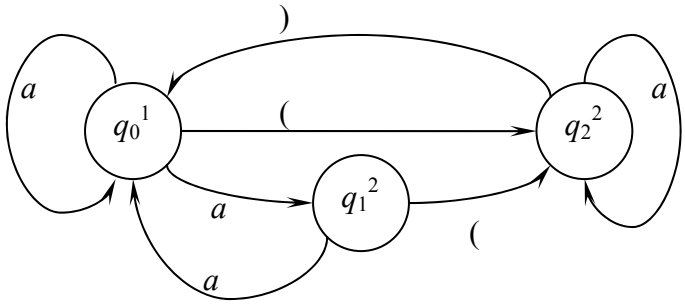


Рисунок 1.12 – Граф ДГКА

20 Сформулируйте теоремы о неразрешимых свойствах стандартных схем программ.

Тесты проверки усвоения материала

1 Функциональные выражения в схемах программ называют:

а) тестами;

б) термами;

в) предикатами;

г) операторами;

д) атрибутами.

2 Термом в схеме программы является слово:

а) $f(a)+g(h(x, y))$;

б) $(2*x-(y+3))/2$;

в) $f(5, g(h(x, y)))$;

г) $f(a, g(h(x, y)))$;

д) $f(x, g(h(x, 3)))$.

3 Логические выражения в схемах программ могут содержать:

а) +, - , *, /;

б) 0, 1, 2, 3, …;

в) 0 и 1;

г) >, <, =, <=, >=.

д) $\wedge$, $\vee$, $\neg$.

4 Инструкция вида l: $x:=\tau$ на l_1 соответствует в графовой форме схемы программы:

а) вершине-преобразователю;

б) вершине-распознавателю;

в) вершине-петле;

г) начальной вершине;

д) конечной вершине.

5 Оператором пересылки в схемах программ является:

а) $x:=z$;

б) $x:=b$;

в) $x:=x+y$;

г) $x:=5$;

д) $x:=x+5$.

6 Схема программы, у которой на каждой дуге заданы все переменные, называется:

а) тотальной;

б) полной;

в) стандартной;

г) пустой;

д) правильной.

7 Пара (S, I), где S - схема в базисе ß, а I - интерпретация этого базиса, называется:

а) памятью схемы;

б) стандартной программой;

в) конфигурацией схемы;

г) состоянием памяти схемы;

д) протоколом интерпретированной схемы программы.

8 Протокол выполнения программы описывается конечной или бесконечной последовательностью:

а) меток;

б) состояний памяти;

в) интерпретаций;

г) конфигураций;

д) цепочек операторов.

9 Если в протоколе выполнения программы $l_{i+1} = l_i$ и $W_{i+1} = W_i$, то оператор схемы в вершине с меткой l_i является:

а) начальным оператором;

б) оператором присваивания;

в) условным оператором;
г) оператором петли;
д) конечным оператором.

10 Если вершина схемы программы с меткой l_i помечена оператором присваивания, а выходящая из нее дуга ведет к вершине с меткой k, то в протоколе:
а) $l_{i+1} = k$ и $W_{i+1} = W_i$;
б) $l_{i+1} = k$, $W_{i+1} \overset{x}{=} W_i$;
в) $l_{i+1} = l_k$, $W_{i+1} \overset{x}{=} W_i$;
г) $l_{i+1} = l_i$ и $W_{i+1} = W_i$;
д) $l_{i+1} = l_k$ и $W_{i+1} = W_i$.

11 Если обе схемы программы зацикливаются либо останавливаются с одинаковым результатом, то они:
а) функционально эквивалентны;
б) лт-эквивалентны;
в) согласованы;
г) корректно эквивалентны;
д) тотально эквивалентны.

12 Различные свободные интерпретации схем программ отличаются только интерпретацией:
а) специальных символов;
б) функциональных символов;
в) переменных;
г) констант;
д) предикатных символов.

13 Результат подстановки терма $p(x, y)[a/x, h(h(x))/y]$:
а) $p(x, h(h(y)))$;
б) $p(x, h(h(a)))$;
в) $p(a, h(h(x)))$;
г) $p(a/x, h(h(x))/y)$;
д) $p(a, h(h(y)))$.

14 Множество логико-термальных историй всех цепочек стандартной схемы, завершающихся заключительным оператором, называют:
а) детерминантом схемы;
б) протоколом программы;
в) языком схемы;
г) интерпретацией схемы;
д) протоколом схемы.

15 Для стандартных схем программ частично разрешимы проблемы:
а) функциональной эквивалентности;
б) пустоты и свободы;
в) свободы;
г) тотальности;
д) пустоты и тотальности.

16 Двоичное слово $\alpha = b_1 b_2 \ldots b_n$ согласовано со свободной интерпретацией базиса ß, если для любого i, $1 \le i \le n$, $I(p)('f^i \alpha\,') =$
а) 0;
б) α;
в) b_i;
г) 1;
д) b_1.

17 Если в ДГКА входной алфавит $T=\{k, l, m, n\}$, то в ДДГКА символу n будет соответствовать код:
а) 00000;
б) 11111;
в) 00001;
г) 10000;
д) 11110.

18 Любое непустое слово на ленте ДДГКА заканчивается:
а) 11;
б) 00;
в) 01;
г) 10.
д) 0.

19 Стандартная схема программы, моделирующая ДДГКА с n вершинами собирается из:
а) n фрагментов;
б) n+4 фрагментов;
в) n+2 фрагментов;
г) n+1 фрагментов.
д) n-1 фрагментов.

20 Для схемы S_8, представленной на рисунке 1.11, цепочкой является:
а) 0, 1, 2^0, 1, 2^1, 4;
б) 0, 1, 2^1, 3, 1, 2^0, 1, 4;
в) 0, 1, 2^0, 3, 1, 2^1, 4;
г) 0, 1, 2^1, 3, 1, 2^0, 4;
д) 0, 1, 2^1, 3, 2^0, 4.

2 Семантическая теория программ

2.1 Формализация семантики программ

Каждая программа определяется правилами ее формального представления (синтаксисом), правилами интерпретации ее содержательного, смыслового значения (семантикой), а также прагматическими свойствами (практической полезностью).

Задача доказательства синтаксической правильности программы была успешно решена благодаря описанию синтаксиса языков высокого уровня на основе формальных грамматик и разработке соответствующих формальных методов. Она решается автоматически в ходе трансляции программы. При этом на программиста не возлагается никакой дополнительной работы, кроме осмысления диагностических результатов синтаксического анализа и внесения в программу необходимых изменений.

Проблема семантической правильности программы оказалась значительно сложнее, поскольку она в отличие от синтаксического анализа существенно зависит от решаемой задачи и исходных данных.

Изучение семантики языка программирования, т.е. смысла различных конструкций языка, чрезвычайно важно для программирования. Придать точный смысл конструкциям языка, а, следовательно, дать полное формальное описание языка программирования можно лишь на строгой математической основе. Такое описание необходимо не только создателям языков программирования, но и разработчикам трансляторов, системным и прикладным программистам на этапах разработки, отладки и документирования программ. Формальное представление семантики языка программирования может служить следующим целям:

- единой и строгой интерпретации смысла языковых конструкций;
- разработке структурированных надежных программ методом пошаговой детализации с одновременным доказательством их корректности;
- автоматизации доказательства семантической правильности программ (автоматизации верификации).

Исследования по формализации семантики начались с середины 1960-х годов, вслед за успехами в формализации синтаксиса. Выработаны три основных подхода к решению проблемы: операционный, детонационный и аксиоматический /3, 4, 5, 6, 7/.

2.1.1. Операционная семантика

Операционная семантика сводится к описанию смысла программы посредством выполнения ее операторов на реальной или виртуальной машине. Смысл оператора определяется изменениями, произошедшими в состоянии машины после выполнения данного оператора.

Пусть состояние компьютера - это значения всех его регистров и ячеек памяти, в том числе коды условий и регистры состояний. Тогда, чтобы определить семантику команды, следует записать текущее состояние компьютера и выполнить данную команду, а затем изучить новое состояние машины.

Описание операционной семантики операторов языков программирования высокого уровня требует создания реального или виртуального компьютера. Аппаратное обеспечение данного компьютера является чистым интерпретатором его машинного языка. Чистый интерпретатор любого языка программирования может быть создан с помощью программных средств, которые становятся виртуальным компьютером для данного языка. Семантику языка высокого уровня можно описать, используя чистый интерпретатор данного языка. При таком подходе существуют две проблемы. Во-первых, сложность и индивидуальные особенности аппаратного обеспечения компьютера и операционной системы, используемых для запуска чистого интерпретатора, затрудняют понимание происходящих действий. Во-вторых, выполненное таким образом семантическое определение будет доступно только для людей с абсолютно идентичной конфигурацией компьютера.

Этих проблем можно избежать, заменив реальный компьютер виртуальным компьютером низкого уровня. Регистры, память, информация о состоянии и процесс выполнения операторов - все это можно смоделировать, соответствующими программами. Набор команд можно создать так, чтобы семантику каждой отдельной команды было легко понять и описать. Таким образом, машина будет идеализирована и значительно упрощена, что облегчит понимание изменений ее состояния.

Использование операционного метода для полного описания семантики языка программирования *L* требует создания двух компонентов. Во-первых, для преобразования языка *L* в операторы выбранного языка низкого уровня нужен транслятор. Во-вторых, для этого языка низкого уровня необходима виртуальная машина, состояние которой изменяется с помощью команд, полученных при трансляции операторов высокого уровня. Именно изменения состояния этой виртуальной машины определяет смысл данного оператора.

Пример 2.1. Семантику конструкции *for* языка С можно описать в терминах следующих простых команд.

Оператор языка С:	Операционная семантика:
for (*expr*1; *expr*2; *expr*3){	*expr*1
...	*loop*: *if expr*2 = 0 *goto out*
}	…
	*expr*3;
	goto loop
	out:

Человек, читающий подобное описание, является «виртуальным компьютером» и считается способным правильно «выполнить» команды описания и распознать эффект такого «выполнения».

Пример 2.2. В качестве примера низкоуровневого языка, который можно применять для операционной семантики, рассмотрим следующий список операторов, соответствующих простым управляющим операторам типичного языка программирования:

ident:= *var*
ident:= *ident* - 1
goto label
if var relop var goto label

Здесь *relop* - одни из операторов отношений из набора {= , <>, >, <, >=, <=}, *ident* - идентификатор, а *var* - идентификатор или константа. Все эти операторы просты и легки для понимания и реализации.

Пример 2.3. Обобщение приведенных в примере 2.2 операторов присваивания позволяет описывать более общие арифметические выражения и операторы присваивания. Обозначим *bin_op* - бинарный арифметический оператор, а *un_op* - унарный оператор, тогда получим:

ident:= *var bin_op var*
ident:= *un_op var*

Многочисленные арифметические типы данных и автоматическое преобразование типов несколько усложняют это обобщение. Введение небольшого количества новых относительно простых команд позволит описывать семантику массивов, записей, указателей и подпрограмм.

Описание операционной семантики функций рассмотрим на примере системы равенств:

$$\begin{cases} f_1(x_1, x_2, \ldots, x_k) = E_1, \\ f_2(x_1, x_2, \ldots, x_k) = E_2, \\ \ldots \\ f_m(x_1, x_2, \ldots, x_k) = E_m. \end{cases} \qquad (2.1)$$

где $x_1, x_2, \ldots, x_k$ - входные данные;

$f_1, f_2, \ldots, f_m$ - определяемые функции с формальными параметрами;

$E_1, E_2, \ldots, E_m$ – выражения над определяемыми функциями с аргументами, зависящими от входных данных.

Операционная семантика интерпретирует эти равенства как систему подстановок.

Определение 2.1. Под подстановкой $<s, E, \tau>$ терма τ в выражение E вместо символа s (в частности, переменной) будем понимать переписывание выражения E с заменой каждого вхождения в него символа s на выражение τ. Каж-

дое равенство $f_i(x_1, x_2, \ldots, x_k) = E_i$ задает в параметрической форме множество правил подстановок вида:

$$<x_1, x_2, \ldots, x_k; f_i(\tau_1, \tau_2, \ldots, \tau_k) \rightarrow E_i; \tau_1, \tau_2, \ldots, \tau_k> \qquad (2.2)$$

где $\tau_1, \tau_2, \ldots, \tau_k$ - конкретные аргументы (значения или определяющие их выражения) данной функции.

Это правило допускает замену вхождения левой его части в какое-либо выражение на его правую часть.

Интерпретация системы равенств 2.1 для получения значений определяемых функций в рамках операционной семантики производится следующим образом. Пусть задан набор входных данных (аргументов) $d_1, d_2, \ldots, d_k$. На первом шаге осуществляется подстановка этих данных в левые и правые части равенств с выполнением там, где это возможно, предопределенных операций и с выписыванием получаемых в результате этого равенств. На каждом следующем шаге просматриваются правые части полученных равенств. Если правая часть является каким-либо значением, то оно и является значением функции, указанной в левой части этого равенства. В противном случае правая часть является выражением, содержащим вхождения каких-либо определяемых функций с теми или иными наборами аргументов. Если для такого вхождения соответствующая функция с данным набором аргументов имеется в левой части какого-либо из полученных равенств, то либо вместо этого вхождения подставляется вычисленное значение правой части этого равенства, либо не производится никаких изменений. В том же случае, если эта функция с данным набором аргументов не является левой частью никакого из полученных равенств, то формируется (и дописывается к имеющимся) новое равенство. Оно получается из исходного равенства для данной функций подстановкой в него вместо параметров указанных аргументов этой функции. Такие шаги осуществляются до тех пор, пока все определяемые функции не будут иметь вычисленные значения.

Пример 2.4. Рассмотрим систему равенств, определяющую функцию $FACT(3) = n!$

$$\begin{cases} FACT(0) = 1, \\ FACT(n) = n \times FACT(n-1). \end{cases}$$

Для вычисления значения $FACT(3)$ осуществляются следующие 6 шагов.

Шаг 1:

$$\begin{cases} FACT(0) = 1, \\ FACT(3) = 3 \times FACT(2). \end{cases}$$

Шаг 2:

$$\begin{cases} FACT(0) = 1, \\ FACT(3) = 3 \times FACT(2), \\ FACT(2) = 2 \times FACT(1). \end{cases}$$

Шаг 3:

$$\begin{cases} FACT(0) = 1, \\ FACT(3) = 3 \times FACT(2), \\ FACT(2) = 2 \times FACT(1), \\ FACT(1) = 1 \times FACT(0). \end{cases}$$

Шаг 4:

$$\begin{cases} FACT(0)=1, \\ FACT(3)=3\times FACT(2), \\ FACT(2)=2\times FACT(1), \\ FACT(1)=1. \end{cases}$$

Шаг 5:

$$\begin{cases} FACT(0)=1, \\ FACT(3)=3\times FACT(2), \\ FACT(2)=2, \\ FACT(1)=1. \end{cases}$$

Шаг 6:

$$\begin{cases} FACT(0)=1, \\ FACT(3)=6, \\ FACT(2)=2, \\ FACT(1)=1. \end{cases}$$

Первым и самым значительным использованием формальной операционной семантики было описание семантики языка PL/I. Эта абстрактная машина и правила трансляции языка PL/I были названы общим именем Vienna Definition Language (VDL) в честь города, в котором они были созданы корпорацией IBM.

Операционная семантика является эффективной до тех пор, пока описание языка остается простым и неформальным. К сожалению, описание VDL языка PL/I настолько сложно, что практическим целям оно фактически не служит.

Операционная семантика зависит от алгоритмов, а не от математики. Операторы одного языка программирования описываются в терминах операторов другого языка программирования, имеющего более низкий уровень. Этот подход может привести к порочному кругу, когда концепции неявно выражаются через самих себя.

2.1.2 Денотационная семантика

Денотационная семантика опирается на теорию рекурсивных функций. Основной концепцией денотационной семантики является определение для каждой сущности языка некоего математического объекта и некоей функции, отображающей экземпляры этой сущности в экземпляры этого математического объекта. Поскольку объекты определены строго, то они представляют собой точный смысл соответствующих сущностей. Сама идея основана на факте существования строгих методов оперирования математическими объектами, а не конструкциями языков программирования. Сложность использования этого метода заключается в создании объектов и функций отображения. Название метода «денотационная семантика» происходит от английского слова *denote* (обозначать), поскольку математический объект обозначает смысл соответствующей синтаксической сущности.

Сущность денотационного метода рассмотрим на примере простой языковой конструкции - двоичные числа. Синтаксис этих чисел можно описать с помощью БНФ следующими грамматическими правилами:

<двоичное_число> ::= 0 | 1 | <двоичное_число> 0 | <двоичное_число> 1

Для описания двоичных чисел с использованием денотационной семантики и грамматических правил, указанных выше, их фактическое значение связывается с каждым правилом, имеющим в своей правой части один терминальный (основной) символ. Объектами в данном случае являются двоичные числа.

В нашем примере значащие объекты должны связываться с первыми двумя правилами. Остальные два правила являются правилами вычислений, поскольку они объединяют терминальный символ, с которым может ассоциироваться объект, с нетерминальным, который может представлять собой некоторую конструкцию.

Пусть область определения семантических значений объектов представляет собой множество неотрицательных десятичных целых чисел *N*. Это те объекты, которые будем связывать с двоичными числами. Семантическая функция M_b отображает синтаксические объекты в объекты множества *N* согласно указанным выше правилам. Сама функция M_b определяется следующим образом:

M_b('0') = 0, M_b('1')=1
M_b(<двоичное_число> '0') = 2*M_b(<двоичное_число>)
M_b(<двоичное_число> '1') = 2*M_b(<двоичное_число>) + 1

Синтаксические цифры заключены в апострофы, чтобы отличать их от математических цифр. Отношение между этими категориями подобно отношениям между цифрами в кодировке ASCII и математическими цифрами. Когда программа считывает число как строку, то прежде, чем это число сможет использоваться в программе, оно должно быть преобразовано в математическое число.

Пример 2.5. Описание значения десятичных синтаксических литеральных констант с помощью БНФ имеет вид:

<десятичное_число> ::= 0|1|2|3|4|5|6|7|8|9
| <десятичное_число> (0|1|2|3|4|5|6|7|8|9)

Денотационные отображения для этих синтаксических правил будут следующими:

M_d('0') = 0, M_d('1') = 1, ..., M_d('9') = 9
M_d(<десятичное_число> '0') = 10*M_d(<десятичное_число>)
M_d(<десятичное_число> '1') = 10*M_d(<десятичное_число>) + 1
…
M_d(<десятичное_число> '9') = 10*M_d(<десятичное_число>) + 9

Денотационную семантику подобно операционной семантике программы можно определить в терминах изменений состояний идеального компьютера. Для простоты состояния определяются только в терминах значений всех переменных, объявленных в программе. Операционная и денотационная семантики различаются тем, что изменения состояний в операционной семантике определяются запрограммированными алгоритмами, а в денотационной - строгими математическими функциями.

Пусть состояние *s* программы определяется набором упорядоченных пар {<i_1, v_1>, <i_2, v_2>, …, <i_n, v_n>}. Каждый параметр *i* является именем переменной,

а соответствующие параметры *v* являются текущими значениями данных переменных. Любой из параметров *v* может иметь специальное значение *undef*, указывающее, что связанная с ним величина в данный момент не определена.

Пусть *VARMAP* - функция двух параметров: имени переменной и состояния программы. Значение функции *VARMAP*(i_k, *s*) равно v_k.

Большинство семантических функций отображения для программ и программных конструкций отображают состояния в состояния. Эти изменения состояний используются для определения смысла программ и программных конструкций. Такие языковые конструкции, как выражения, отображаются не в состояния, а в величины.

Пример 2.6. Рассмотрим простые выражения, содержащие не более одного оператора + или *, единственными операндами которых являются скалярные переменные и целочисленные литеральные константы; круглые скобки не используются; значения выражений являются целыми числами. Описание этих выражений в форме БНФ имеет вид:

<выражение> ::= <десятичное_число> | <переменная>
| <двоичное_выражение>
<двоичное_выражение> ::= <выражение_слева> <оператор>
<выражение_справа>
<оператор> ::= + | *

Единственной рассматриваемой нами ошибкой в выражениях будет неопределенное значение переменной. Другие ошибки в большинстве случаев зависят от машины. Пусть *Z* - набор целых чисел, а *error* - ошибочное значение. Символ ≡ будет обозначать равенство по определению функции.

Функция отображения для данного выражения *E* и состояния *s* будет записана на псевдокоде следующим образом:

M_E (<выражение>, *s*) ≡
case <выражение> *of*
<десятичное_число>: M_d(<десятичное_число>, *s*)
<переменная>: *if VARMAP*(<переменная>, *s*) = *undef*
then error
else VARMAP(<переменная>, *s*)
<двоичное_выражение>:
if (M_E(<двоичное_выражение>.<выражение_слева**>,** *s*) = *undef or*
M_E(<двоичное_выражение>.<выражение_справа>, *s*) = *undef*)
then error
else if (<двоичное_выражение>.<оператор> = '+'
then M_E(<двоичное_выражение>.<выражение_слева>, *s*) +
M_E(<двоичное_выражение>.<выражение_справа>, *s*)
else M_E(<двоичное_выражение>.<выражение_слева>, *s*)*
M_E(<двоичное_выражение>.<выражение_справа>, *s*)

Денатационную семантику операторов рассмотрим на примере оператора присваивания. Оператор присваивания можно представить как вычисление выражения плюс присваивание его значения переменной, находящейся в левой части. Таким образом, семантика оператора будет описываться на псевдокоде следующей функцией:

$M_a(x = E, s) \equiv$
if $M_E(E, s) = error$ *then error*
else $s' = \{<i_1', v_1'>, <i_2', v_2'>, ..., <i_n', v_n'>\}$ *where*
for $j = 1, 2, ..., n$
  if $i_j <> x$ *then* $v_j' = VARMAP(i_j', s)$ *else* $M_E(E, s)$

При этом сравнение $i_j <> x$ относится к именам, а не значениям переменных.

Определив полную систему функций для заданного языка, можно использовать ее для описания смысла программ на этом языке. Это создает основу для строгого способа мышления в программировании. Денотационная семантика может использоваться для разработки языка. С одной стороны, денотационные описания очень сложны, с другой - они дают хороший метод краткого описания языка.

2.1.3 Аксиоматическая семантика

Наиболее практичным с позиции доказательства семантической правильности программ считается аксиоматический подход, предложенный Хоаром. В нем программа рассматривается как функция, преобразующая начальное состояние памяти в конечное. Свойства операторов языка выражаются в виде логических формул, устанавливающих отношения между переменными программы. В этом случае явно описываются свойства состояний памяти, а не сами состояния. Формальная семантика языка программирования состоит в том, что формулируются некоторые утверждения о свойствах конструкций языка, из которых можно вывести утверждения о свойствах конкретной программы /8/.

Аксиоматическая семантика языка программирования задается некоторым формальным языком утверждений о свойствах программ, а также системой аксиом и правил вывода, описывающих свойства операторов языка программирования. Выводимые в данной аксиоматической системе утверждения характеризуют свойства программ.

Итак, опишем решаемую задачу или функцию, которую должна выполнять программа, т.е. специфицируем свойства программы. Эту функцию (обозначим ее f) надо задать так, чтобы было видно, к каким значениям применима функция и к какой области относятся ее результаты. Обозначим область определения функции f через D, а область значений - через R. Поскольку функция f задает отображение f: $D \rightarrow R$, она представляет собой некоторое подмножество (отношение) $f \subseteq D \times R$. Поэтому функция f характеризуется заданием отображе-

ния (схемы функции) $D \rightarrow R$, области определения D, области значений R и связи между элементом $d \in D$ и элементом $r \in R$. Логическая спецификация определяет функцию f в виде:

- $pre\text{-}f(d)$, $d \in D$ - предусловие функции f, характеризующее множество элементов области D;

- $post\text{-}f(d, r)$, $d \in D$, $r \in R$ - постусловие функции f, характеризующее связь между $d \in D$ и $r \in R$.

Формальный смысл этих предикатов устанавливается их взаимосвязью следующей формулой:

$$\forall\, d(pre\text{-}f(d) \rightarrow post\text{-}f(d, f(d))) \qquad (2.3)$$

Т.е. для всех $d \in D$ истинность предусловия влечет истинность постусловия.

При программной реализации Pg функции f на компьютере абстрактные множества D и R интерпретируются на множестве $State$ состояний памяти. При этом каждое состояние памяти характеризуется множеством значений переменных программы Pg, включая и неопределенные.

Предусловие и постусловие программы должны определяться при условии завершения вычислений, т.е. достижения выхода программы. Дело в том, что при некоторых начальных состояниях памяти выполнение программы может зациклиться либо привести к неопределенному результату. Ясно, что такие состояния не могут входить в область определения нашей функции f, и, следовательно, она является не полностью определенной (частичной). Теперь логическая спецификация программы Pg, описывающая ее внешнее функционирование, может быть определена на множестве состояний памяти предикатами:

$$pre\text{-}Pg: State \rightarrow Boolean,$$
$$post\text{-}Pg: State \times State \rightarrow Boolean.$$

При условии:

$$\forall\, st(pre\text{-}Pg(st) \wedge fin(Pg, st) \rightarrow post\text{-}Pg(st, f(st))),$$

где $st \in State$ - начальное (входное) состояние программы Pg;
$f(st) \in State$ - конечное (выходное) состояние программы Pg;
$fin(Pg, st)$ - предикат завершен выполнения программы Pg.

Предикаты $pre\text{-}Pg$ и $post\text{-}Pg$, характеризуют только вход и выход программы. Для более полной логической спецификации свойств программы обычно требуется характеризовать ее поведение в некоторых промежуточных, точках. Для этой цели используют так называемые инварианты.

Определение 2.2. Инвариантом точки q программы называют формулу $\varphi_q(x_1, x_2, \ldots, x_m)$ логического языка спецификации, такую, что φ_q истинно для на-

бора значений $x_1, x_2,\ldots, x_m$ программных переменных при каждом прохождении выполнения через точку q.

Инвариантность утверждения означает независимость его истинности от пути, по которому достигается заданная точка. Обычно точку связывают с некоторым сечением управляющего графа программы (схемы алгоритма решения задачи).

Понятие инварианта применимо, в частности, к началу и концу программы. Здесь инварианты задают предусловие и постусловие, характеризуя область исходных данных и целевую функцию программы. В промежуточных точках инварианты характеризуют некоторые подцели, достижимые в процессе вычислений.

Для формализации свойств отдельного оператора A программы с позиций воздействия его на состояние памяти обычно используют тройку Хоара $\{P\}A\{Q\}$, где P и Q - инварианты входа и выхода оператора A. Естественна трактовка P и Q как предусловия и постусловия оператора A. Смысл тройки Хоара: если P истинно перед выполнением оператора A и выполнение оператора A завершается, то Q истинно для значений переменных, соответствующих завершению A. Формально тройка Хоара представляется предикатом:

$$\{P\}A\{Q\}: \forall st(P(st) \wedge fin(A, st) \rightarrow Q(st, f_A(st))), \tag{2.4}$$

где f_A - функция, вычисляемая оператором A.

Тройка Хоара является утверждением об операторе и представляет собой элементарную единицу рассуждений о свойствах программы в аксиоматической теории. Она формально описывает семантику оператора. Всю программу можно рассматривать как один сложный оператор и связать с ней тройку Хоара. Тогда, как следует из формулы (2.4), доказательство частичной корректности программы сводится к доказательству истинности ее тройки Хоара.

Пример 2.7. Истинные тройки Хоара для вариантов операторов присваивания и цикла:

а) $\{x>0\}\ x:=x+1\ \{x>1\}$;

б) $\{x>0\}$ *while* $x>0$ *do* $x:=x-1$; $\{x=0\}$.

Важным общезначимым свойством тройки Хоара является возможность изменения предусловия или постусловия фиксированного оператора A с сохранением истинности его тройки. Соответствующие правила изменения (усиления и ослабления) установлены законами логического следствия (консеквенции):

$$\left.\begin{array}{l} \{P\}\ A\ \{R\}, R \rightarrow Q \mid - \{P\}\ A\ \{Q\} \\ P \rightarrow R, \{R\}\ A\ \{Q\} \mid - \{P\}\ A\ \{Q\} \end{array}\right\} \tag{2.5}$$

Первый закон утверждает, что если выполнение оператора A обеспечивает истинность постусловия R, то оно также обеспечивает истинность каждого постусловия Q, которое следует из R. Второй закон утверждает, что если R - из-

вестное предусловие оператора A, приводящего к постусловию Q, то это же относится к любому другому утверждению P, которое влечет R. Покажем в качестве примера справедливость: первого закона. По определению его посылки истинны, т.е. $\{P\}A\{R\} \equiv t$ и $R \rightarrow Q \equiv t$. Отсюда: $P \equiv t$ и $R \equiv t$. Теперь предположим, что заключение ложно, т.е. $\{P\}A\{Q\} \equiv f$. Тогда $Q \equiv f$. Но это значит, что $R \rightarrow Q \equiv f$, что неверно. Значит, наше предположение о том, что $\{P\}A\{Q\} \equiv f$ неверно, что и требовалось доказать.

Говорят, что Q в первом из рассматриваемых законов является ослаблением R, а P во втором - усилением R. Таким образом, законы консеквенции утверждают, что для любого оператора A можно произвольным образом ослаблять (вплоть до *true*) постусловие и произвольным образом усиливать (вплоть до *false*) предусловие. При этом истинное значение тройки Хоара сохраняется. Отсюда следует неизменная истинность троек Хоара вида:

$$\{P\}A\{t\} \text{ и } \{f\}A\{Q\}.$$

Определения троек Хоара для простых операторов играют роль аксиом, а для композиций операторов устанавливаются правила вывода, выражающие свойства композиции через свойства ее составных частей. Рассмотрим принципы формализации семантики на примере некоторых типовых операторов языка Паскаль, не вдаваясь в детали модификации правил для их применения в конкретных алгоритмах верификации.

Пустой оператор не производит никакого действия и не меняет состояние памяти. Следовательно, его предусловие и постусловие одинаковы, и его семантику можно описать тройкой $\{P\}\{P\}$ или тройкой $\{Q\}\{Q\}$.

Оператор присваивания вида $x := e$, где x - переменная некоторого типа; e - выражение того же типа. Его семантика выражается тройкой:

$$\{Q(x \leftarrow e)\}x := e\ \{Q\} \qquad (2.6)$$

Эта аксиома утверждает, что если постусловие Q истинно при подстановке в него e вместо x, то Q должно быть истинно после присваивания переменной x значения e. Иногда вместо записи $Q(x \leftarrow e)$ применяют эквивалентное ей обозначение Q_e^x. Очевидно, семантику оператора присваивания можно также описать тройкой $\{P\}\ x := e\ \{P(x \rightarrow e)\}$.

Пример 2.8. Определение истинных предусловий и постусловий для заданных операторов присваивания:

а) $\{P\}\ x := 5\{x = 5\}$, $P \equiv 5 = 5 \equiv t$;

б) $\{a+b = 2\}\ x := a+b\ \{Q\}$, $Q \equiv (x=2)$;

в) $\{P\}\ x := x*x\ \{x^4 = 256\}$, $P \equiv ((x*x)^4 = 256) \equiv (x^8 = 256)$;

г) $\{x \geq y\}\ x := x-y\ \{Q\}$, $Q \equiv (x \geq 0)$;

д) $\{P\}\ x := a + b\{y = 2\}$, $P \equiv (y = 2)$.

Последний вариант иллюстрирует формализацию важного свойства оператора присваивания: выполнение присваивания одной переменной не может изменить значение другой переменной.

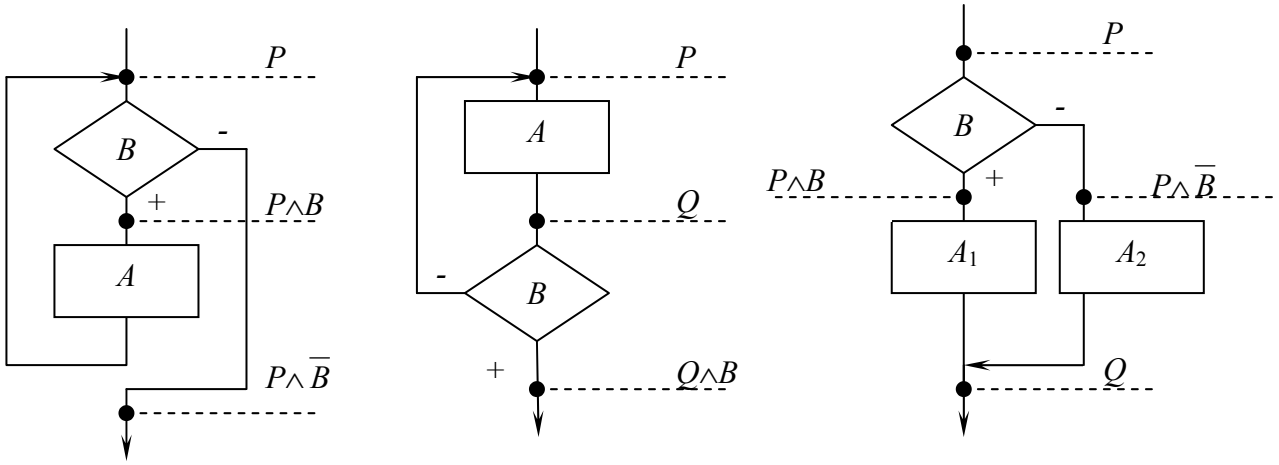
Семантика сложных операторов описывается правилами вывода, показывающими, как свойства составной конструкции связаны со свойствами ее частей. Каждое правило содержит одну посылку или несколько и единственное следствие (тройку Хоара для рассматриваемого сложного оператора).

Семантика **составного оператора** характеризуется тем, что свойства в неизменной форме могут переноситься из постусловия одного оператора последовательности в предусловие следующего за ним оператора. Правило вывода для последовательности операторов формулируется так: из истинности троек Хоара операторов последовательности выводится (следует) истинность тройки Хоара последовательности операторов. Любую последовательность операторов можно рассматривать как последовательность двух операторов. Правило вывода для последовательности двух операторов имеет вид:

$$\{P\}\ A_1\ \{V\},\ \{V\}\ A_2\ \{Q\}\ |\text{-}\ \{P\}\ A_1; A_2\ \{Q\}. \qquad (2.7)$$

Если требуется установить истинность спецификации **условного оператора** *if B then* A_1 *else* A_2, то необходимо, как это видно из рисунка 2.1 в, доказать два утверждения относительно истинности спецификаций альтернативных операторов:

$$\{P \wedge B\}\ A_1\ \{Q\},\ \{P \wedge \overline{B}\}\ A_2\ \{Q\}\ |\text{-}\{P\}\ \textit{if}\ B\ \textit{then}\ A_1\ \textit{else}\ A_2\ \{Q\}. \qquad (2.8)$$



а) цикл с предусловием б) цикл с постусловием в) ветвление

Рисунок 2.1 - Схемы циклов и ветвления

Пример 2.9. Доказательство истинности тройки Хоара для условного оператора $\{t\}$ *if* $a>0$ *then* $m:=a$ *else* $m:=-a$ $\{m\geq 0\}$.

Согласно правилу вывода (2.8) для решения задачи необходимо показать истинность двух троек:

$$\{t \wedge a>0\}\ m:=a\ \{m\geq 0\} \text{ и } \{t \wedge \overline{a>0}\}\ m:=-a\ \{m\geq 0\}.$$

Докажем истинность первой из этих троек, для чего воспользуемся аксиомой (2.6) и вычислим истинное предусловие оператора $m:=a$ по заданному

постусловию $m \geq 0$. Запишем истинную тройку с полученным предусловием: $\{a \geq 0\}\ m := a\ \{m \geq 0\}$. Теперь попробуем усилить в этой тройке предусловие, воспользовавшись законом консеквенции (2.5):

$$t \wedge a > 0 \rightarrow a \geq 0,\ \{a \geq 0\}\ m := a\ \{m \geq 0\} \vdash \{t \wedge a > 0\}\ m := a\ \{m \geq 0\}.$$

Для этого осталось только показать истинность первой посылки:

$$t \wedge a > 0 \rightarrow a \geq 0 \equiv \overline{t \wedge a > 0} \vee (a \geq 0) \equiv f \vee \overline{a > 0} \vee (a > 0) \vee (a = 0) \equiv t.$$

Аналогично доказывается истинность тройки для оператора $m := -a$. Здесь доказательство также сводится к демонстрации истинности первой посылки:

$$t \wedge \overline{a > 0} \rightarrow -a \geq 0 \equiv \overline{t \wedge \overline{a > 0}} \vee (a \leq 0) \equiv f \vee (a > 0) \vee (a < 0) \vee (a = 0) \equiv t.$$

Таким образом, заданная тройка Хоара для условного оператора истинна.

Истинность тройки Хоара для **оператора цикла** *while B do A* выводится из истинности тройки Хоара для его тела *A*. При этом истинность тройки для *A* должна сохраняться при каждом выполнении *A*. Из этих соображений и схемы цикла на рисунке 2.1 а строим правило вывода:

$$\{P \wedge B)\ A\ \{P\} \vdash \{P\}\ \textit{while } B \textit{ do } A\ \{P \vee \overline{B}\} \qquad (2.9)$$

где *P* - предусловие оператора цикла, являющееся инвариантом.

Подобные рассуждения с привлечением схемы цикла на рисунке 2.1 б приводят к правилу вывода для тройки Хоара оператора *repeat A until B*:

$$\{P\}\ A\ \{Q\},\ (Q \vee \neg B) \rightarrow P \vdash \{P\}\ \textit{repeat } A \textit{ until } B\ \{Q \wedge B\} \qquad (2.10)$$

где *Q* - инвариант цикла.

Подобным образом можно построить правила вывода для любых операторов и аксиоматическую систему для любого языка программирования в целом.

2.2 Верификация программ

2.2.1 Методы доказательства правильности программ

Определение 2.3. Свойство программы, характеризующее отсутствие в ней ошибок по отношению к целям разработки, называют корректностью (правильностью) программы.

В силу многих причин программист не может гарантировать правильность своей программы. Правильность программы, как формальную, так и смысловую, нужно доказать. К неформальным методам доказательства правильности программ относят отладку и тестирование. Тестирование – это процесс выполнения программы с целью найти ошибки. Выполнить всестороннее тестирование сложной программы практически невозможно. Обычно в ходе тестирования программы исследуются некоторые пути вычислений, реализуемые при выбранных вариантах исходных данных. Тестирование не позволяет доказать корректность программы. Поэтому корректность часто оценивают вероятностной характеристикой - надежностью программы. Грамотное тестирование в комплексе с отладочными действиями повышает надежность программы.

Корректность программы - относительное понятие: оно имеет смысл только для фиксированной и четко сформулированной цели разработки программы.

Определение 2.4. Формализованное описание постановки задачи называют спецификацией задачи.

Определение 2.5. Верификация – это процесс доказательства соответствия между программной реализацией задачи и спецификацией задачи.

Цель верификации - демонстрация свойства корректности программы. Если тестирование ограничено исследованием отдельных реализаций (прогонов) программы, то верификация охватывает все допустимые ее реализации. При этом в отличие от тестирования верификация представляет собой аналитическое исследование свойств программы по ее тексту, т.е. без выполнения самой программы. Следовательно, цель верификации может быть достигнута только строгим математическим доказательством соответствия программы и спецификации. В качестве базы для проведения таких доказательств необходимо построить формальную систему для описания свойств корректности программы.

Корректность программы характеризуют двумя свойствами:

1) частичной корректностью - удовлетворением спецификации задачи при условии завершения программы, т.е. достижения выхода;

2) завершением - достижением выхода программы в ходе выполнения, начатого в допустимом состоянии.

2.2.2 Доказательство частичной корректности программы методом индуктивных утверждений

Наиболее распространенным методом доказательства частичной корректности программ является метод индуктивных утверждений. Он позволяет свести анализ свойств программы к доказательству конечного числа утверждений, записанных в виде формул логического языка спецификации и имеющих интерпретацию в проблемной области решаемой задачи.

Рассмотрим метод применительно к программам, оперирующим с конечным множеством простых переменных $\{x_1, x_2,\ldots, x_n\}$, где $n\geq 1$. Ограничимся для простоты только двумя типами выполняемых операторов языка программиро-

вания: оператором присваивания вида $x := f(x_1, x_2, \ldots, x_n)$ и условным оператором вида *if* $\alpha(x_1, x_2, \ldots, x_n)$ *then* L_+ *else* L_-, где операторы L_+ и L_- соответствуют истинному (знак +) и ложному (знак минус) значениям предиката α.

Зададим спецификацию программы *Pg*, т.е. формально укажем, каким требованиям должно удовлетворять ее выполнение. Для этого записывают соответствующие инвариантные формулы на языке логической спецификации, связывая их с входом, выходом и некоторыми промежуточными (контрольными) точками программы:

$$inv_{t1}(x_1, x_2, \ldots, x_n): P;\ inv_{t2}(x_1, x_2, \ldots, x_n): Q;\ inv_{t3}(x_1, x_2, \ldots, x_n), \ldots$$

где t_1 - точка входа в программу (предусловие программы);
t_2 - точка выхода из программы (постусловие программы);
t_3 ,... - контрольные точки инвариантов циклов.

Эти формулы являются утверждениями о том, что должно быть неизменно истинно в выбранных точках программы. Они являются опорными (контрольными) соотношениями при доказательстве корректности программы.

Определение 2.6. Программу, снабженную инвариантными утверждениями (в форме комментариев), называют аннотированной.

Теперь программу можно рассматривать как тройку Хоара $\{P\}Pg\{Q\}$, в истинности которой нужно убедиться.

При выполнении программы с различными вариантами исходных данных возможна реализация различных цепочек операторов. Эти цепочки имеют общие начало и конец. Назовем такие цепочки трассами вычислений. Истинность тройки Хоара $\{P\}Pg\{Q\}$ имеет место, если истинны тройки Хоара $\{P_j\}T_j\{Q\}$ для всех трасс T_j, т.е.

$$\{P\}Pg\{Q\}: \forall j(\{P_j\}T_j\{Q\}).$$

Трассы вычислений разбивают предусловие *P* таким образом, что

$$P = \vee P_j;\ \forall j_1, j_2\ ((j_1 \neq j_2) \rightarrow P_{j1} \wedge P_{j2} = false).$$

Т.е. предусловие хотя бы одной трассы имеет значение истина, и предусловия любой пары трасс не будут истинны одновременно. При наличии в программе повторяющихся вычислений перебор всех трасс обычно оказывается невозможен. Эту проблему устраняют введением инварианта цикла - утверждения, приписанного циклу, которое должно сохранять истинное значение при каждом выполнении тела цикла. В этом случае достаточно проанализировать только одно прохождение по циклу.

Введем понятие условия пути α (обозначим U_α) между соседними контрольными точками схемы алгоритма программы *Pg*. Пусть путь α ведет от некоторой точки *r* к точке *t* и составляет последовательность действий $A_1, \ldots, A_i$,

..., A_k, где A_i, - присваивание или предикат ρ_ε. Значение $\varepsilon \in \{+, -\}$ указывает истинность (+) или ложность (-) предиката ρ. Определим условие пути α следующим образом:

$$U_\alpha: inv_r(x_1, x_2, \ldots, x_n) \rightarrow U_0 \qquad (2.11)$$

где U_0 - индуктивное утверждение, которое выводится методом обратной подстановки (по индукции) из U_k по следующим правилам.

1) U_k: $inv_t(x_1, \ldots, x_n)$;
2) для i от k до 1 выполнить:
а) если A_i - оператор присваивания вида $x_s := f(x_1, x_2, \ldots, x_n)$, то

$$U_{i-1}: U_i(x_s \leftarrow f(x_1, x_2, \ldots, x_n));$$

б) если A_i - предикат ρ_ε, то

$$U_{i-1}: \rho \rightarrow U_i \text{ при } \varepsilon = + \text{ или } U_{i-1}: \neg\rho \rightarrow U_i \text{ при } \varepsilon = -.$$

Запись 2а означает, что индуктивное утверждение U_{i-1} получается подстановкой в U_i, выражения $f(x_1, x_2, \ldots, x_n)$ из правой части оператора присваивания вместо переменной x_s левой части этого оператора. Правило 2б требует построить логическое выражение U_{i-1} по известному выражению U_i для предиката ρ оператора программы. Для истинного значения предиката выражение U_{i-1} связывает операцией импликации ρ и U_i, для ложного значения предиката в выражении используется $\neg\rho$.

Пример 2.10. Определение условия пути α между точками r и t (рисунок 2.2) при заданных предусловии P: $inv_r(x)$ и постусловии Q: $inv_t(x)$ этого пути.

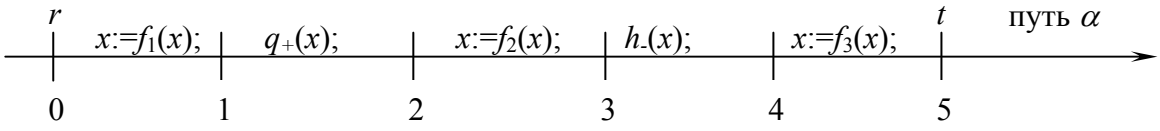


Рисунок 2.2 - Путь α между точками r и t

1) Вывод U_0 по правилам математической индукции:

U_5: $Q(x)$;
U_4: $Q(f_3(x))$;
U_3: $(\neg h(x) \rightarrow Q(f_3(x)))$;
U_2: $(\neg h(f_2(x)) \rightarrow Q(f_3(f_2(x))))$;
U_1: $(g(x) \rightarrow (\neg h(f_2(x)) \rightarrow Q(f_3(f_2(x)))))$;
U_0: $(g(f_1(x)) \rightarrow (\neg h(f_2(f_1(x))) \rightarrow Q(f_3(f_2(f_1(x))))))$.

2) Условие пути U_α будет иметь вид:

$U_\alpha : P \rightarrow (g(f_1(x)) \rightarrow (\neg h(f_2(f_1(x))) \rightarrow Q(f_3(f_2(f_1(x))))))$.

Введение понятия условия пути U_α между соседними контрольными точками позволяет свести задачу анализа частичной корректности программы Pg к доказательству истинности условий U_α между соседними контрольными точками. Условия U_α называются условиями корректности (или условиями верификации). Они являются формулами логического языка спецификации, и их набор для всех путей между соседними контрольными точками полностью характеризует свойство частичной корректности программы. Возможности их доказательства зависят от аксиоматической теории проблемной области, в которой интерпретируются эти формулы.

2.2.3 Методика верификации программы

Доказательство правильности программы «вручную», основанное на применении метода индуктивных утверждений, содержит следующие основные этапы:

1) построение схемы алгоритма решения задачи;

2) формулировка утверждений для входа и выхода программы (предусловия и постусловие) в виде формул логического языка спецификации;

3) выявление всех циклов и формулировка контрольного утверждения (инварианта цикла) для каждого из них на логическом языке спецификации;

4) составление списка путей между контрольными точками алгоритма;

5) построение условия верификации для каждого пути с использованием семантики операторов, образующих путь;

6) Доказательство истинности всех условий верификации как теорем формальной теории проблемной области решаемой задачи;

7) доказательство завершения программы.

Проиллюстрируем изложенную методику на классическом примере.

Пример 2.11. Доказать частичную корректность программы вычисления частного q и остатка r от деления целых неотрицательных чисел x и y.

1 Построим схему алгоритма (рисунок 2.3) и выделим на ней контрольные точки на входе программы (A_1), на выходе программы (A_2) и на входе в цикл (A_3).

2 Запишем предусловие P и постусловие Q программы (утверждения inv_{A1} и inv_{A2} в точках A_1 и A_2). Предусловие определяем, исходя из того, что делимое x и делитель y не должны быть отрицательными, а делитель, кроме того, не принимает нулевого значения. Когда вычисления заканчиваются, остаток r должен быть меньше делителя y и делимое x будет равно сумме остатка с произведением делителя на частное q. Эти требования сформулируем в качестве постусловия. Таким образом:

P: $inv_{A1}(x, y) \equiv ((x \geq 0) \wedge (y > 0))$;
Q: $inv_{A2}(x, y, q, r) \equiv ((x = r + y \times q) \wedge (r < y))$.

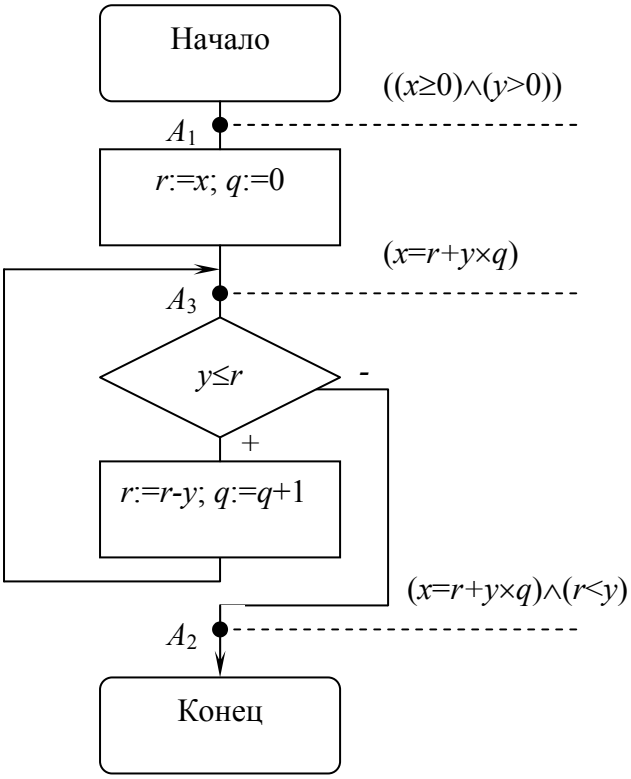


Рисунок 2.3 - Схема алгоритма с контрольными точками

3 В алгоритме есть единственный цикл. При выборе инварианта цикла примем во внимание, что сумма остатка r и произведения делителя и частного q, вычисленная для текущих значений r и q, при каждом прохождении цикла должна быть неизменна и равна делимому:

$$inv_{A3}(x, y, q, r) \equiv (x = r+y\times q).$$

4 Список путей между контрольными точками алгоритма содержит три элемента:

а) путь α_1 от A_1 к A_3;

б) путь α_2 от A_3 к A_3 через тело цикла;

в) путь α_3 от A_3 к A_2 через условие цикла.

5 Построим условия верификации $U_{\alpha 1}$, $U_{\alpha 2}$ и $U_{\alpha 3}$ для выделенных путей программы.

а) путь α_1:

U_1: $inv_{A3}(x, y, q, r) \equiv (x=r+y\times q)$;

U_0: $inv_{A3}(x, y, q\leftarrow 0, r\leftarrow x) \equiv (x=x+y\times 0)$;

$U_{\alpha 1}$: $((x\geq 0)\wedge(y>0))\rightarrow(x=x+y\times 0)$.

б) путь α_2:

U_2: $inv_{A3}(x, y, q, r) \equiv (x=r+y\times q)$;

U_1: $inv_{A3}(x, y, q\leftarrow q+1, r\leftarrow r-y)$;

U_0: $(y\leq r)\rightarrow inv_{A3}(x, y, q\leftarrow q+1, r\leftarrow r-y) \equiv ((y\leq r)\rightarrow(x=r+y\times q))$;

$U_{\alpha 2}$: $(x=r+y\times q)\rightarrow((y\leq r)\rightarrow(x= r+y\times q))$.

в) путь α_3:

U_1: $inv_{A2}(x, y, q, r) \equiv ((x=r+y\times q) \wedge (r<y))$;
U_0: $\neg(y\le r) \to inv_{A2}(x, y, q, r) \equiv (\neg(y\le r) \to ((x=r+y\times q) \wedge (r<y)))$;
$U_{\alpha 3}$: $(x=r+y\times q) \to (\neg(y\le r) \to ((x=r+y\times q) \wedge (r<y)))$.

6 Докажем истинность условий верификации. Используя известные аксиомы арифметики простых переменных, перепишем теоремы, сформулированные в пункте 5, следующим образом:

$U_{\alpha 1}$: $((x\ge 0) \wedge (y>0)) \to t$;
$U_{\alpha 2}$: $(x=r+y\times q) \to ((y\le r) \to (x=r+y\times q))$;
$U_{\alpha 3}$: $(x= r+y\times q) \to ((y>r) \to ((x=r+y\times q) \wedge (y>r)))$.

Все условия верификации имеют вид $a \to b$, и обычный метод доказательства таких теорем сводится к тому, чтобы показать, что либо b истинно при истинном a, либо b всегда истинно, либо a всегда ложно, либо a и b одновременно истинны (ложны). Первое из трех условий верификации истинно, поскольку второй операнд операции импликации есть всегда истина. При доказательстве второй и третьей теорем удобно применить первый и второго закон для импликации и закон де Моргана. Для удобства преобразований примем: $a = (x=r+y\times q)$, $c = (y\le r)$, $d = (y>r)$. С учетом этих обозначений выводим:

$U_{\alpha 2}$: $a \to (c \to a) \equiv (a \wedge c) \to a \equiv \neg(a \wedge c) \vee a \equiv (\neg a \vee \neg c) \vee a \equiv a \vee (\neg a \vee \neg c) \equiv$
$\equiv (a \vee \neg a) \vee \neg c \equiv t \vee \neg c \equiv t$;
$U_{\alpha 3}$: $a \to (d \to (a \wedge d)) \equiv ((a \wedge d) \to (a \wedge d)) \equiv t$.

Таким образом, частичная корректность программы доказана.

2.2.4 Анализ завершения программы

Для доказательства общей корректности программы необходимо помимо свойства частичной корректности показать свойство завершения этой программы.

Программа с заданным предусловием P может не обладать свойством завершения по крайней мере по двум причинам:

1) обращение к частично определенной функции со значением аргумента вне области его определения (например, при делении на нуль);

2) выполнение бесконечной последовательности операторов (например, зацикливание при выполнении вычислений итеративного характера).

Первая проблема традиционно решается квалифицированными разработчиками программно, путем контроля значений аргументов частично определенных функций. Поэтому ограничимся рассмотрением анализа завершения только циклических вычислений.

Для решения проблемы логического анализа завершения циклических программ используют ограничивающие функции. Ограничивающая функция подбирается в каждом конкретном случае так, чтобы она характеризовала конечность повторяющегося вычислительного процесса. Рассмотрим, как определяется ограничивающая функция для операторов цикла.

Для цикла с верхним окончанием (например, *while B do A*) ограничивающая функция *t* представляет собой выражение с целочисленными значениями. Поведение (семантика) функции *t* формально определяется так:

$$\begin{cases} P \wedge B \to t > 0, \\ \forall\, t_0\ (\{0 < t = t_0\}\ A\ \{0 \le t < t_0\}) \end{cases}, \tag{2.12}$$

где *P* - предусловие оператора цикла.

Схема цикла (рисунок 2.1 а) поясняет содержательный смысл ограничивающей функции: перед телом цикла *A* должно быть всегда справедливо утверждение $t>0$, а при каждом выполнении тела цикла (т.е. для каждого целого t_0) значение *t* уменьшается оставаясь всегда неотрицательным. Величина, на которую изменяется *t* зависит от решаемой задачи и определяется телом цикла. Удовлетворение функции *t* условиям (2.12) обеспечивает конечность циклического процесса.

Для оператора цикла с нижним окончанием (например, *repeat A until B*), которому соответствует схема на рисунке 2.1 б, ограничивающая функция *t* формально определяется формулой:

$$\begin{cases} P \to t > 0, \\ \forall\, t_0\ (\{0 < t = t_0\}\ A\ \{0 \le t < t_0\}). \end{cases} \tag{2.13}$$

Причина отличия объясняется чем, что предусловием оператора *A* в данном случае будет *P*, а не $P \wedge B$. Доказав частичную корректность цикла, достаточно показать истинность утверждений (2.12) или (2.13), чтобы сделать заключение об общей корректности цикла.

Пример 2.12. Доказать завершение программы деления целых чисел (смотри пример 2.7). В качестве ограничивающей функции целесообразно выбрать выражение $r+y$. Такой выбор обусловлен существованием инварианта $r+y>0$ единственного цикла алгоритма, так как в цикле всегда справедливо $r\ge 0$ и $y>0$. Кроме того, значение выражения $r+y$ при каждом выполнении тела цикла ($r:=r-y$; $q:=q+1$) уменьшается. В алгоритме использован цикл с верхним окончанием. Таким образом, формально ограничивающая функция имеет семантику:

$$\begin{cases} (x = r + y*q) \wedge (y \le r) \to (r + y > 0), \\ \forall t_0\ (\{0 < r + y = t_0\}\ r := r - y;\ q := q + 1\ \{0 < r + y < t_0\}). \end{cases}$$

Покажем истинность первого утверждения. Из $y>0$ следует $y\le r \equiv r\ge y \equiv r>0 \equiv r+y>0$. Введем обозначения: $a = (x = r + y*q)$ и $b = (r + y > 0)$, тогда имеем:

$$(x = r + y*q) \wedge (y \le r) \to (r + y > 0) \equiv (x = r + y*q) \wedge (r + y > 0) \to (r + y > 0) \equiv a \wedge b \to a \equiv \neg(a \wedge b) \vee a \equiv (\neg a \vee \neg b) \vee a \equiv (\neg a \vee a) \vee \neg b \equiv t \vee \neg b \equiv t.$$

Значит, первое из двух утверждений справедливо.

В истинности второго утверждения убеждаемся, применяя, например, к постусловию ($0 < r+y < t_0$) преобразующие действия операторов тела цикла, в надежде получить предусловие. Использованное здесь преобразование постусловия Q в предусловие P принято обозначать $P = Q(r \leftarrow r-y)$. Иногда более удобным может быть преобразование $Q = P(r-y \leftarrow r)$. При этом принимается во внимание цепочка очевидно эквивалентных выражений:

$$(0<r+y<t_0)(r \leftarrow r-y) \equiv (0<r<t_0) \equiv (y<r+y<t_0+y) \equiv (0<r+y=t_0).$$

Данное доказательство носит индуктивный характер, поэтому t_0 рассматривается как новое значение перед каждым выполнением тела цикла. Максимальное значение $t_0 = x+y$ ограничивающая функция $r+y$ получает только перед первым выполнением тела цикла.

Таким образом, доказана завершаемость программы, а с учетом ранее доказанной ее частичной корректности и полная корректность.

2.2.5 Анализ незавершения программы

Если не удается подобрать ограничивающую функцию для доказательства завершения программы, это не означает, что она обязательно не будет завершаться. Незавершение программы нужно попытаться доказать.

Пример 2.13. Доказать незавершение аннотированного фрагмента программы вычисления наибольшего общего делителя:

```
begin {(x>0) ∧ (y>0)}
    a: = x; b: = y; {(a>0) ∧ (b>0)}
    repeat
        while a≥b do a: = a - b;
        {0≤a<b}
        while b>a do b: = b - a;
        {0<b≤a}
   until a = b
end.
```

Проанализируем текст программы. Второй цикл *while-do* может начаться при $a = 0$ и тогда не завершится. Теперь найдем причину появления значения $a = 0$. Оно проявляется в первом цикле *while-do* как следствие кратности значения a значению b перед входом в цикл ($a=k*b$, где $k \geq 1$). Таким образом, если x и y удовлетворяют условию $(x>0) \wedge (y>0) \wedge \exists k\ ((k \geq 1) \wedge (x=k*y))$, то рассматриваемая программа не будет завершаться.

Доказательство незавершения полезно тем, что оно позволяет найти причину незавершения и устранить ошибку. В рассматриваемом примере для исправления программы достаточно поменять условие $a \geq b$ на $a>b$ в заголовке первого из циклов *while-do*.Тогда контрольное соотношение после этого цикла будет иметь вид $0 < a \leq b$.

2.3 Автоматизация верификации программ

Верификация программ - трудоемкий процесс, и его целесообразно автоматизировать. Проблемы корректности программы алгоритмически неразрешимы, поэтому не следует рассчитывать на полную автоматизацию. Не могут быть автоматическими аннотирование программы и доказательство условий верификации. Синтез инвариантов циклов можно автоматизировать, однако на практике этого не делают из-за ограниченности разработанных методов. Генерацию условий верификации можно выполнять автоматически. В связи с этим применение компьютеров для верификации программ в настоящее время идет преимущественно по пути создания автоматизированных систем верификации, предусматривающих диалог с пользователем на некоторых этапах работы. Для краткости будем называть такую программу верификатором. Типовая схема верификатора показана на рисунке 2.4.

На вход системы поступает подготовленная пользователем аннотированная программа. В ней, в частности, предусмотрены формулы логического языка спецификации для входного и выходного условий программы, а также инвариант для каждого цикла.

В ходе анализа программы выполняется контроль лексической и синтаксической правильности аннотированной программы подобно тому, как это делается в трансляторах. Программа переводится на промежуточный язык. В качестве такого языка обычно используется одна из форм бесскобочной записи. Анализ аннотированной программы выполняется автоматически.

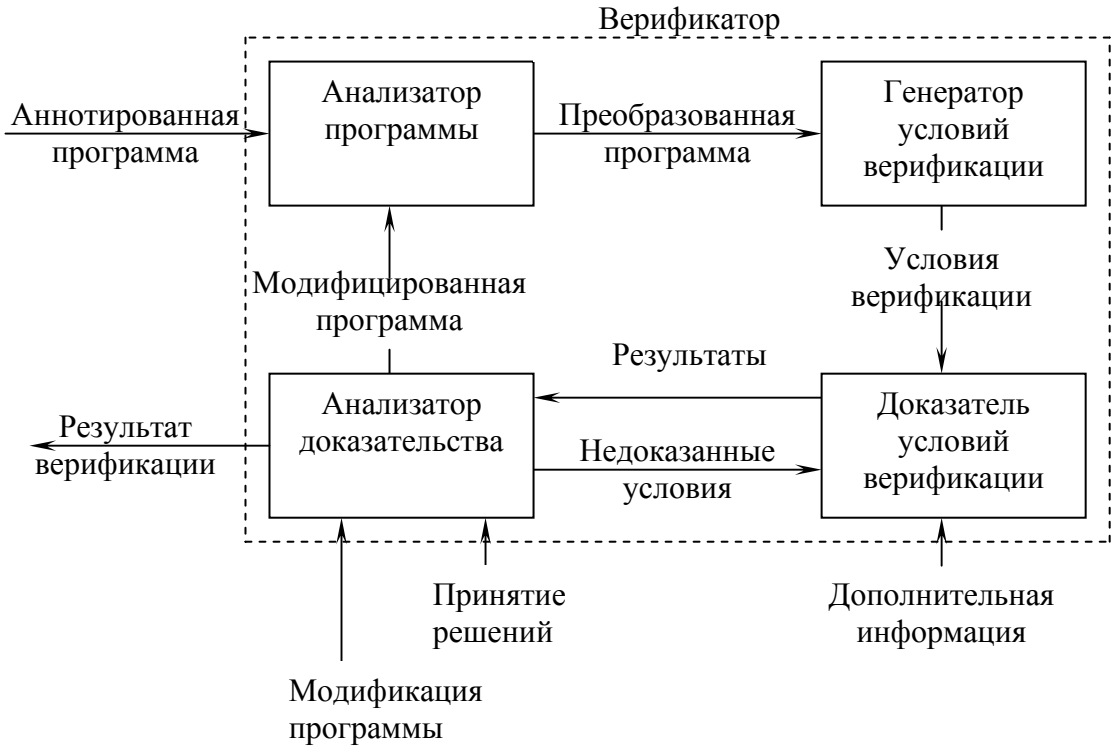


Рисунок 2.4 – Типовая схема верификатора

Генерация условий верификации основана на применении аксиоматической системы используемого языка программирования и также осуществляется без участия пользователя. Имеются два альтернативных алгоритма генерации условий верификации: алгоритм прямого прослеживания и алгоритм обратного прослеживания. Алгоритм прямого прослеживания строит формулы условий, просматривая программу от начала к концу. При этом в соответствии с семантикой операторов программы их предусловия преобразуются в постусловия. Обратное прослеживание происходит от конца программы к ее началу и предполагает последовательное преобразование постусловий операторов в их предусловия. Естественно, правила вывода, образующие аксиоматическую систему одного и того же языка программирования, будут различны для разных алгоритмов прослеживания.

При доказательстве условий верификации могут выполняться некоторые эквивалентные преобразования и упрощения условий верификации. Например, арифметические и логические выражения приводятся к канонической форме, определенные части выражений замещаются логическими константами и т.д. Каноническая форма арифметического выражения - это упорядоченная сумма произведений. Логическое выражение представляется как последовательность конъюнкций более простых выражений, не содержащих других логических операций, кроме отрицания. В некоторых случаях элементарные упрощения позволяют убедиться в истинности отдельных условий верификации.

На этом этапе могут также вычисляться истинностные значения тех частей условий верификации, которые сформулированы в рамках достаточно простых аксиоматических теорий. Подстановка результатов таких вычислений в условия верификации позволяет упростить их, а в ряде случаев доказать или опровергнуть. При необходимости возможно применение системы аксиом пользователя.

Результаты доказательства условий верификации исследуются анализатором доказательства. Он может обнаружить следующие ситуации.

1 Все условия верификации истинны. Работа завершается.

2 Доказательство отдельных условий верификации не завершено. Такие условия возвращаются на доказательство с применением дополнительной информации (аксиом пользователя), вводимой пользователем в ходе работы верификатора.

3 Среди условий верификации обнаружены ложные. В этом случае ошибки могут быть как в спецификации программы, так и в операторах самой программы, причем формально разделить эти ситуации невозможно. Пользователь может выполнить модификацию аннотированной программы и повторить всю процедуру ее обработки.

Принятие окончательных решений в ходе анализа верификатором результатов доказательства возлагается на пользователя.

Вопросы и задания для самоконтроля

1 Для чего необходимо формальное представление семантики языка программирования?

2 Перечислите известные подходы к формализованному описанию семантики языков программирования.

3 Каким образом описывается операционная семантика операторов языков программирования?

4 Опишите семантику оператора *while* языка программирования *Pascal* с помощью операционной семантики.

5 Что является основной концепцией денотационной семантики?

6 Опишите вещественные числа с использованием денотационной семантики.

7 Чем задается аксиоматическая семантика языка программирования?

8 Запишите формальную спецификацию программы *Pg* на множестве состояний памяти *State* в аксиоматической семантике.

9 Дайте определение инварианта контрольной точки программы. Приведите пример.

10 Как формально описываются семантические свойства отдельного оператора программы?

11 В чем смысл тройки Хоара? Приведите пример истинных троек Хоара для операторов присваивания, условия и цикла.

12 Сформулируйте законы консеквенции.

13 Опишите формально семантику типовых операторов какого-либо языка программирования.

14 Определите истинные предусловия и постусловия для заданных операторов присваивания языка программирования *Pascal*:

а) $\{P\}$ x:=(a+b)/2 $\{x = 4\}$;

б) $\{a\text{-}b = 3\}$ x: = *sqr*(a-b) $\{Q\}$/

15 Докажите истинность тройки Хоара условного оператора $\{(a>0)\wedge(b>0)\}$ *if* $a>b$ *then* m:=a-b *else* m:=b-a $\{m\geq 0\}$.

16 В чем разница между тестированием и верификацией программы?

17 Какими свойствами характеризуется корректность программы?

18 Запишите правила построений условий путей между соседними контрольными точками программы по методу математический индукции.

19 Составьте условие пути β, заданного рисунком 2.5 при заданном предусловии $P(x, y)\equiv (x\geq 0)\wedge(y\geq 0)$ и постусловии $Q(x, y)\equiv (x<0)\wedge(y<0)$.

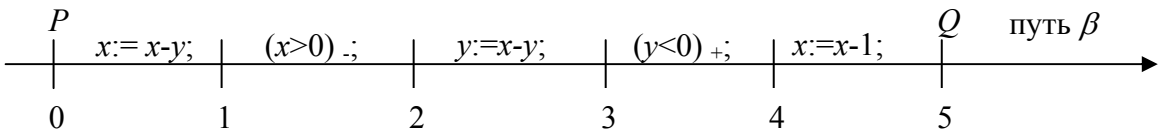


Рисунок 2.5 - Путь между соседними контрольными точками β

20 Сформулируйте этапы верификации программы на основе индуктивных утверждений.

21 Докажите полную корректность программы, вычисляющей наименьшее общее кратное двух целых неотрицательных чисел без использования деления и умножения.

22 Опишите семантику ограничивающей функции для циклов с верхним и нижним окончанием.

23 Изобразите и поясните типовую схему верификатора.

Тесты проверки усвоения материала

1 Правила интерпретации смыслового значения программы называются:

а) синтаксисом;

б) семантикой;

в) прагматикой;

г) формализацией;

д) спецификацией.

2 Семантика, сводящаяся к описанию смысла программы посредством выполнения ее операторов на реальной или виртуальной машине, называется:

а) операционной;

б) аксиоматической;

в) денатационной;

г) декларативной;

д) императивной.

3 В основе операционной семантики операторов языков программирования лежат:

а) строгие методы оперирования математическими объектами;

б) семантические функции отображения для программных конструкций;

в) система аксиом и правил вывода;

г) реальный или виртуальный компьютер;

д) язык логической спецификации.

4 В операционной семантике каждое равенство $f_i(x_1, x_2, \dots, x_k) = E_i$ задает в параметрической форме множество правил подстановок вида:

а) $<\tau_1, \tau_2, \dots, \tau_k; f_i(x_1, x_2, \dots, x_k) \rightarrow E_i; \tau_1, \tau_2, \dots, \tau_k>$

б) $<E_i; \tau_1, \tau_2, \dots, \tau_k \rightarrow x_1, x_2, \dots, x_k; f_i(\tau_1, \tau_2, \dots, \tau_k)>$

в) $<E_i; f_i(\tau_1, \tau_2, \dots, \tau_k) \rightarrow x_1, x_2, \dots, x_k; \tau_1, \tau_2, \dots, \tau_k>$

г) $<x_1, x_2, \dots, x_k; \tau_1, \tau_2, \dots, \tau_k \rightarrow E_i; f_i(\tau_1, \tau_2, \dots, \tau_k)>$

д) $<x_1, x_2, \dots, x_k; f_i(\tau_1, \tau_2, \dots, \tau_k) \rightarrow E_i; \tau_1, \tau_2, \dots, \tau_k>$.

5 Первым использованием формальной операционной семантики было описание семантики языка:

а) Fortran;

б) Pascal;

в) PL/I;

г) Алгол;

д) Ada.

6 Денотационная семантика каждой сущности языка сопоставляет:

а) состояние виртуальной машины;

б) математический объект;

в) предикат;

г) правило вывода;

д) логическую спецификацию.

7 Денотационное отображение для синтаксического правила <десятичное_число> ::= <десятичное_число> 5 имеет вид:

а) M_d(<десятичное_число> '5') = 10*M_d(<десятичное_число>) + 5;

б) M_d(<десятичное_число> '5') = 5*M_d(<десятичное_число>) + 10;

в) M_d(<десятичное_число> '5') = M_d(<десятичное_число>) + 5;

г) M_d(<десятичное_число> '5') = 5*M_d(<десятичное_число>);

д) M_d(<десятичное_число> '5') = M_d^{10}(<десятичное_число>) + 5.

8 Предикат, описывающий предусловие программы *Pg* в аксиоматической семантике на множестве состояний памяти *State*:

а) *pre-Pg*: *State* → *State*;

б) *pre-Pg*: *Boolean* → *State*;

в) *pre-Pg*: *State* → *Boolean*;

г) *pre-Pg*: *State* →*Boolean* × *Boolean*;

д) *pre-Pg*: *Boolean* ×*Boolean* → *State*.

9 Для характеристики поведения программы в некоторых промежуточных точках в аксиоматической семантике используются:

а) инварианты;

б) тройки Хоара;

в) аксиомы;

г) правила вывода;

д) ограничивающие функции.

10 Истинная тройка Хоара для оператора цикла:
а) $\{x\geq 0\}$ *for* $i:=1$ *to* 5 *do* $x:=x+i$; $\{x\geq 5\}$;
б) $\{x\geq 0\}$ *for* $i:=1$ *to* 5 *do* $x:=x+i$; $\{x>15\}$;
в) $\{x\geq 0\}$ *for* $i:=1$ *to* 5 *do* $x:=x*i$; $\{x\geq 15\}$;
г) $\{x\geq 0\}$ *for* $i:=1$ *to* 5 *do* $x:=x*i$; $\{x\geq 5\}$;
д) $\{x\geq 0\}$ *for* $i:=1$ *to* 5 *do* $x:=x+i$; $\{x\geq 15\}$.

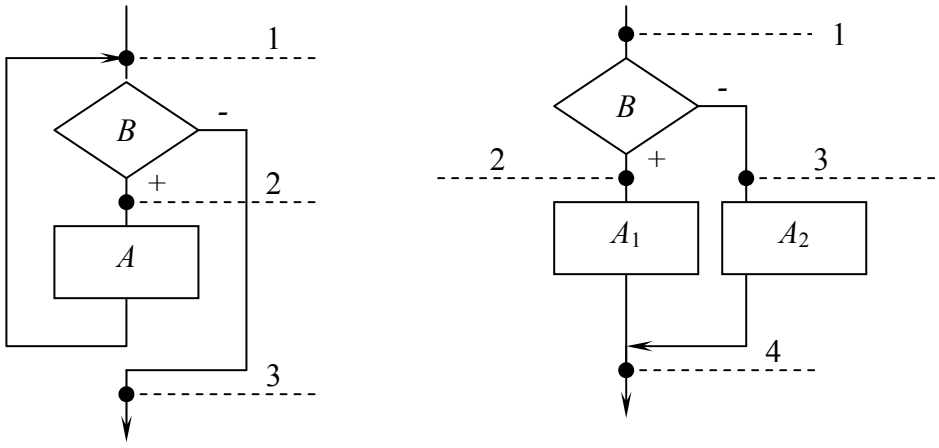
11 Тождественно истинная тройка Хоара:
а) $\{t\}A\{Q\}$;
б) $\{P\}A\{f\}$;
в) $\{P\}A\{P\}$;
г) $\{P\}A\{t\}$;
д) $\{Q\}A\{Q\}$.

12 Семантика оператора присваивания задается аксиомой:
а) $\{Q(x \leftarrow e)\}x:=e\ \{Q\}$;
б) $\{Q(x \rightarrow e)\}x:=e\ \{Q\}$;
в) $\{P\}\ x:=e\ \{P(x \rightarrow \text{e})\}$;
г) $\{P\}\ x:=e\ \{Q(x \leftarrow \text{e})\}$;
д) $\{P(x \rightarrow e)\}x:=e\ \{Q\}$.

13 Для оператора $\{x < y\}\ x:=x-y\ \{Q\}$ истинным постусловием будет:
а) $x\leq 0$;
б) $x>0$;
в) $x<0$;
г) $x\geq 0$;
д) $x>y$.

14 Правило вывода для оператора цикла с заданным предусловием P и условием цикла B имеет вид:
а) $\{P \wedge B)\ A\ \{P\}$ |- $\{P\}$ *while B do A* $\{P \vee \overline{B}\}$;
б) $\{P\}\ A\ \{\overline{B}\}$ |- $\{P\}$ *while B do A* $\{P \vee \overline{B}\}$;
в) $\{P \wedge B\}\ A\ \{P\}$ |- $\{P\}$ *while B do A* $\{P \vee B\}$;
г) $\{P \wedge \overline{B}\}\ A\ \{P\}$ |- $\{P\}$ *while B do A* $\{P \vee B\}$;
д) $\{\overline{B}\}\ A\ \{P\}$ |- $\{P\}$ *while B do A* $\{P \vee B\}$.

15 При заданном предусловии P и постусловии Q оператора ветвления, представленного на рисунке 2.6 б, точке 3 будет соответствовать условие:
а) P;
б) $P \wedge B$;
в) $P \wedge \overline{B}$;
г) Q;
д) $Q \wedge B$.



а) цикл б) ветвление

Рисунок 2.6 – Схемы операторов

16 При заданном предусловии P и постусловии Q оператора ветвления, представленного на рисунке 2.6 б, условие $P \wedge B$ будет соответствовать точке:

а) 1;
б) 2;
в) 3;
г) 4;
д) 2 и 4.

17 При заданном предусловии P и постусловии Q оператора цикла, представленного на рисунке 2.6 а, инвариантом точки 2 будет условие:

а) P;
б) $P \wedge B$;
в) $P \wedge \overline{B}$;
г) Q;
д) $Q \wedge B$.

18 Для пути γ, заданного на рисунке 2.7, условие пути будет иметь вид:

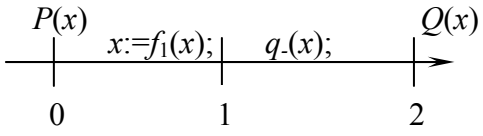


Рисунок 2.7 – Путь γ

а) $P(x) \rightarrow (q(f_1(x)) \rightarrow Q(f_1(x)))$;
б) $P(x) \rightarrow (\neg q(x) \rightarrow Q(x))$;
в) $P(x) \rightarrow (\neg q(f_1(x)) \wedge Q(f_1(x)))$;
г) $P(x) \rightarrow (q(f_1(x)) \wedge Q(f_1(x)))$;
д) $P(x) \rightarrow (\neg q(f_1(x)) \rightarrow Q(f_1(x)))$.

19 Полной автоматизации подлежит следующий этап верификатора:

а) анализ доказательства;

б) доказательство условий верификации;

в) генерация условий верификации;

г) аннотирование программы;

д) принятие решения.

20 Синтаксическую правильность программы в ходе автоматической верификации проверяет:

а) анализатор программы;

б) генератор условий верификации;

в) доказатель условий верификации;

г) анализатор доказательств;

д) пользователь.

3 Асинхронные процессы

3.1 Формальное определение асинхронного процесса

Системы, обладающие конечным числом состояний, называются дискретными системами. Примерами дискретных систем служат ЭВМ, их элементы, устройства, сети ЭВМ, программы и операционные системы, системы автоматизированного управления объектами и процессами. Существенными чертами таких систем являются:

1) фазовость – наличие у процессов явно выраженных фаз;

2) согласованность – переход к следующей фазе процесса только после получения сигнала об окончании предыдущего перехода;

3) параллельность – возможность одновременных переходов в нескольких подпроцессах;

4) асинхронность – отказ от времени и тактированных последовательностей изменений состояний процесса и замена их на причинно-следственные связи.

Определим основные понятия и формализуем асинхронный процесс согласно В.И. Варшавскому /9/.

Ситуация (от лат. *situatio* – положение) – сочетание условий, создающих определенную обстановку. В каждый момент процесс может находиться только в одной ситуации.

Процесс (от лат. *processus* - продолжение) – описание динамики смены ситуаций.

Рассмотрим некоторое множество ситуаций процесса $S = \{S_1, S_2, \ldots\}$ и два подмножества этого множества:

1) $I \subset S$ – подмножество ситуаций, активизирующих процесс, элементы которого будем называть инициаторами;

2) $R \subset S$ – подмножество финальных ситуаций процесса, элементы которого будем называть результантами.

На множестве $S \times S$ введем два отношения:

1) F – отношение непосредственного следования ситуаций процесса; запись вида $S_i F S_j$ означает, что за i-той ситуацией процесса непосредственно следует j-тая ситуация;

2) M – отношение возможности перехода от одной ситуации к другой; запись вида $S_i M S_j$ означает, что из i-той ситуации процесса достижима j-тая ситуация.

Отношение M можно выразить через степень отношения F. Запись вида $S_i F^n S_j$ означает, что существует n-1 промежуточных ситуаций S_a, S_b, …, S_ω таких, что $S_i F S_a$, $S_a F S_b$, …, $S_\omega F S_j$. Т.е. существует траектория, включающая n+1 вершину и n дуг, ведущая из S_i в S_j.

Отношения F и M можно изобразить графом, вершины которого соответствуют ситуациям процесса, а дуга ведет из вершины S_i в вершину S_j, если и только если $S_i F S_j$.

Пример 3.1. Для процесса P_1, граф которого показан на рисунке 3.1 справедливы следующие соотношения: $S_1 F S_2$; $S_1 F S_3$; $S_2 F S_4$; $S_1 M S_4$; $S_1 M S_5$; и т.д.

Определение 3.1. Асинхронным процессом (АП) называется четверка вида $<S, F, I, R>$. Где:

1) S - непустое множество ситуаций;

2) F - отношение непосредственного следования ситуаций, определенное на множестве $S \times S$ ($F \subset S \times S$);

3) I - множество инициаторов ($I \subset S$), т.е. таких ситуаций из S, которые не могут быть только следствием, а обязательно должны быть причиной, удовлетворяющих условию:

если iFs_k, $i \in I$, $s_k \notin I$, то из $s_k F s_l$ следует $s_l \notin I$.

4) R – множество результантов ($R \subset S$). При этом, если какая либо ситуация объявлена результантом, то любая другая, следующая за ней ситуация, если существует, тоже является результантом, т.е.

если rFs, $r \in R$, то $s \in R$.

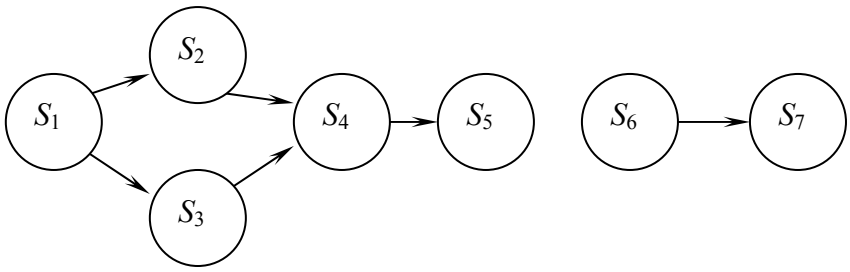


Рисунок 3.1 – Отношение F процесса P_1

Пример 3.2. Рассмотрим процесс P_1, граф которого показан на рисунке 3.1. Выбирая различные наборы инициаторов и результантов, получаем:

а) если $I = \{S_1, S_4, S_6\}$; $R = \{S_5, S_7\}$, то процесс P_1 – не асинхронный, т.к. нарушено условие для инициаторов;

б) если $I = \{S_1, S_6\}$; $R = \{S_4, S_7\}$, то процесс P_1 – не асинхронный, т.к. нарушено условие для результантов;

в) если $I = \{S_1\}$; $R = \{S_5, S_7\}$, то процесс P_1 – асинхронный.

Определение 3.2. Если ситуации s_i и s_k некоторого асинхронного процесса связаны отношением $s_i M s_k$ ($s_i F^n s_k$), то фрагмент процесса, содержащий все части траекторий, ведущих из s_i в s_k, назовем переходом s_i - s_k.

Пусть имеется последовательность ситуаций $S(1)$, $S(2)$, …, $S(i)$, $S(i+1)$, …, где $S(i)$ – элемент, стоящий в последовательности на i-том месте. Для АП можно составить множество последовательностей ситуаций, в которых для каждого места i, кроме последнего, справедливо $S(i)FS(i+1)$ и таких, что ни одна из последовательностей не является частью другой. Все такие последовательности называются допустимыми. Каждая допустимая последовательность ситуаций описывает возможный ход процесса смены ситуаций (траекторию АП) и соответствует реализации АП.

3.2 Подклассы асинхронных процессов

Дадим определение некоторым частным видам АП.

Определение 3.3. Дескрептивным АП называется четверка вида $<S, \Phi, I, R>$. Где:

1) S - непустое множество ситуаций;

2) Φ – бинарное отношение на $P(S)$ (множестве подмножеств ситуаций), которое некоторому подмножеству ситуаций α ставит в соответствие подмножество ситуаций β, т.е. $\alpha \Phi \beta$;

3) I - множество инициаторов ($I \subset P(S)$) таких, что выполняются условия:

а) $(\forall\, i \in I)\ (\exists\, \alpha \in P(S))$: $i \Phi \alpha$;

б) если $i \Phi \alpha$ и $\alpha \notin I$, то из $\alpha \Phi \beta$ следует $\beta \notin I$;

в) $(\forall\, i \in I)\ (\exists\, \alpha \in I)$: $\alpha \Phi i$;

4) R – множество результантов ($R \subset P(S)$) таких, что выполняются условия:

а) если $(\forall\, r \in R)$ имеет место $r \Phi \alpha$, то следует $\alpha \in R$;

б) $(\forall\, r \in R)\ (\alpha \in P(S))$: $\alpha \Phi r$.

Определение 3.4. АП называется автономным, если множество инициаторов и результантов пусто, т.е. $I = \varnothing$ и $R = \varnothing$.

Определение 3.5. АП называется эффективным, если выполняются условия:

1) $(\forall s \in S \backslash R)\ (\exists\, r \in R)$: sMr;

2) $(\forall s \in S \backslash I)\ (\exists\, i \in I)$: iMs;

3) $\neg\, (\exists\, s_i, s_j)$: $((s_i \notin R) \wedge (s_j \notin R) \wedge (s_i M s_j) \wedge (s_j M s_i))$.

Другими словами, для эффективности асинхронного процесса необходимо, чтобы:

1) все траектории графа процесса заканчивались в результантах;

2) все траектории графа процесса начинались в инициаторах;

3) все циклы графа состояли только из результантов.

Пример 3.3. Рассмотрим процесс P_2, граф которого показан на рисунке 3.2. Выбирая различные наборы инициаторов и результантов, получаем:

а) если $I = \{S_1, S_8, S_9\}$; $R = \{S_4, S_5, S_6, S_7, S_{11}\}$, то процесс P_2 – не асинхронный, т.к. нарушено первое условие (для результантов);

б) если $I = \{S_1, S_9\}$; $R = \{S_3, S_4, S_5, S_6, S_7, S_{11}\}$, то процесс P_1 – асинхронный, но не эффективный, т.к. нарушено второе условие (для инициаторов);

в) если $I = \{S_1, S_8, S_9\}$; $R = \{S_3, S_4, S_5, S_6, S_7\}$, то процесс P_1 –асинхронный, но не эффективный, т.к. нарушено первое условие (для результантов);

г) если $I = \{S_1, S_8, S_9\}$; $R = \{S_5, S_6, S_7, S_{11}\}$, то процесс P_1 – асинхронный, но не эффективный, т.к. нарушено третье условие (для циклов);

д) если $I = \{S_1, S_8, S_9\}$; $R = \{S_3, S_4, S_5, S_6, S_7, S_{11}\}$, то процесс P_1 – асинхронный и эффективный.

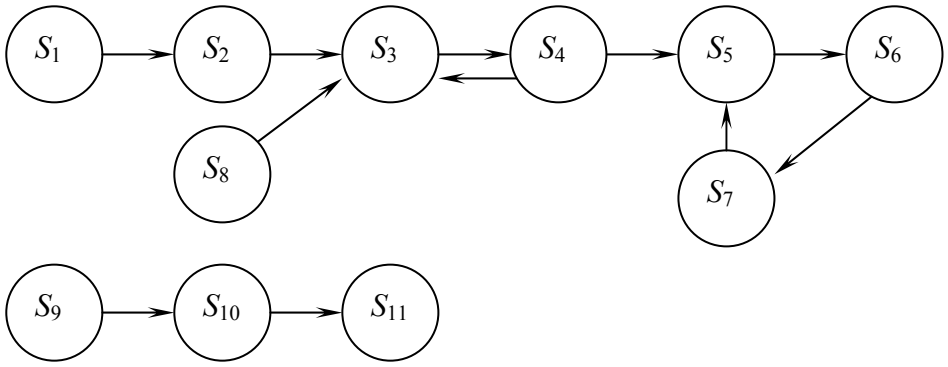


Рисунок 3.2 - Отношение F процесса P_2

Определим на множестве S отношение эквивалентности E следующим образом:

1) $(\forall s \in S)\ sEs$;

2) $(\forall s_i, s_j \in S)\ s_iEs_j \Leftrightarrow (s_iMs_j) \wedge (s_jMs_i)$.

Отношение E произведет разбиение множества S на классы эквивалентности $\pi = \{S(1), S(2), \ldots, S(p)\}$. Для классов эквивалентности определим отношение непосредственного следования за классом Φ. Допустимыми последовательностями классов эквивалентности называются последовательности классов, в которых соседние классы связаны отношением Φ. Тогда справедливы следующие утверждения:

1) если ситуация $S \in S(j)$ является инициатором, то все ситуации классов $S(1), \ldots, S(j)$ являются инициаторами;

2) если ситуация $S \in S(j)$ является результантом, то все ситуации классов $S(j), S(j+1), \ldots, S(q)$, где $S(q)$ – заключительной класс, являются результантами;

3) для эффективного АП любой начальный класс состоит только из инициаторов, а любой заключительный класс состоит только из результантов;

4) для эффективного АП любой класс эквивалентности ситуаций, не принадлежащих к результантам, состоит из одной ситуации.

Определение 3.6. Эффективный АП называется управляемым, если любая допустимая последовательность классов эквивалентности ведет из одного начального класса в один и только один заключительный класс.

Пример 3.4. Рассмотрим эффективный асинхронный процесс P_2 (случай д), граф которого показан на рисунке 3.2. Для проверки процесса на управляемость построим граф отношения непосредственного следования для классов эквивалентности (рисунок 3.3).

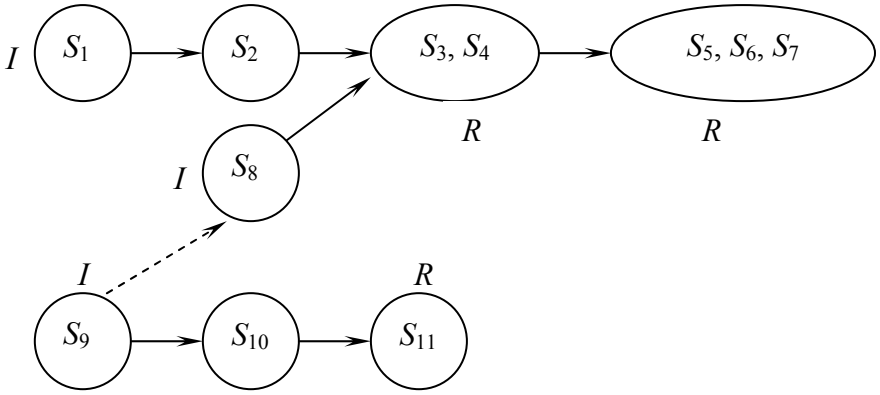


Рисунок 3.3 – Отношение непосредственного следования для классов эквивалентности

Данный процесс является управляемым. Но если отношение Φ дополнить новой парой $S_9 \Phi S_8$ (пунктир на рисунке), то процесс становится не управляемым.

Определение 3.7. Эффективный асинхронный процесс называется простым, если выполняются условия:

1) $(\forall i \in I)\ (\forall s \in S)\ iFs \Rightarrow s \notin I$;

2) $(\forall s \in S)\ (\forall r \in R)\ sFr \Rightarrow s \notin R$.

Пример 3.5. Рассмотрим процесс P_1, граф которого показан на рисунке 3.1. Выбирая различные наборы инициаторов и результантов, получаем:

а) если $I = \{S_1, S_6\}$; $R = \{S_4, S_5, S_7\}$, то процесс P_1 – асинхронный, эффективный, но не простой, т.к. нарушено второе условие (для результантов);

б) если $I = \{S_1, S_2, S_6\}$; $R = \{S_5, S_7\}$, то процесс P_1 – асинхронный, эффективный, но не простой, т.к. нарушено первой условие (для инициаторов);

в) если $I = \{S_1, S_6\}$; $R = \{S_5, S_7\}$, то процесс P_1 – асинхронный, эффективный, управляемый, простой.

Определение 3.8. Протоколом простого АП называется отношение $Q \subset I \times R$.

Протокол простого АП можно рассмотреть как простой АП, в котором за инициатором непосредственно следует результант. В протоколе множество ситуаций $S = I \cup R$, а для каждой пары (i, r) пучки траекторий, ведущих из i в r, заменяются одной дугой.

3.3 Структурирование асинхронного процесса

Структурирование состояний системы необходимо для уточнения понятия «ситуация». Структурирование может быть выполнено различными способами. Одним из них является разбиение ситуации на компоненты (события). При этом каждой компоненте ставится в соответствие предикат p_i, принимающий значение 1, если соответствующее ему условие истинно, и 0 в противном случае. Ситуация при таком структурировании представляется двоичным вектором, размерность которого равна числу выделенных компонент, т.е. $S_i = (p_1, p_2, \ldots, p_n)$, где $p_j \in \{0, 1\}$, $1 \le j \le n$.

Пример 3.6. Выполним моделирование процесса последовательной обработки запросов сервером базы данных. Пусть сервер находится в состоянии ожидания до тех пор, пока от пользователя не поступит запрос клиента, который он обрабатывает и отправляет результат обработки пользователю. Выделим в этой системе события и поставим им в соответствие предикаты:

1) предикат $p_1 = 1$, если запрос клиента поступил;

2) предикат $p_2 = 1$, если сервер начинает обработку запроса;

3) предикат $p_3 = 1$, если сервер заканчивает обработку запроса;

4) предикат $p_4 = 1$, если результат обработки отправляется клиенту.

При таком структурировании ситуации процесса будут иметь вид: $S_1 = (0,0, 0, 0)$, $S_2 = (1, 1, 0, 0)$, $S_3 = (1, 0, 1, 0)$, $S_4 = (1, 0, 0, 1)$.

3.4 Интерпретация асинхронного процесса

3.4.1 Асинхронный процесс как метамодель

Асинхронный процесс – это обобщенная модель, порождающая различные динамические модели. Порождение частных моделей предусматривает интерпретацию асинхронных процессов, которая выполняется в два этапа.

Этап 1. Модельная интерпретация. Состоит в том, что основным понятиям асинхронного процесса (ситуациям, инициаторам, результантам, отношению следования) ставятся в соответствие понятия частных моделей. При этом система соответствий базируется на введении дополнительных ограничений на АП.

Примерами модельных интерпретаций АП могут быть модель Маллера (см. раздел 3.4.2), сигнальный граф (см. раздел 3.4.3) и сети Петри (см. раздел 7).

Этап 2. Предметная интерпретация. Состоит в том, что асинхронному процессу или его модельным интерпретациям при решении конкретных задач ставятся в соответствие некоторые метки, обозначающие символы, операторы, атрибуты или предикаты из заданного множества. Данная интерпретация зависит от специфики решаемых задач.

Примером предметной интерпретации АП может быть моделирование процесса последовательной обработки запросов сервером базы данных (пример 3.7).

3.4.2 Диаграммы переходов

Рассмотрим асинхронный процесс, обладающий следующими свойствами.

1 Ситуации из S структурированы, т.е. представлены в виде наборов (a_1, a_2, ..., a_n) значений двоичных переменных z_i, $1 \le i \le n$.

2 Отношение F изображается орграфом, вершины которого соответствуют ситуациям. Если s_iFs_j, то вершины s_i и s_j соединены дугой, направленной из s_i в s_j, причем если в s_i компонента $z_k=a_i$, а в s_j компонента $z_k=a_j$ и $a_i \neq a_j$, a_i, $a_j \in \{0,1\}$, то в наборе s_i значение a_i компоненты z_k помечается звездочкой. Компоненты, значения которых помечены звездочкой, называются возбужденными, остальные - устойчивыми.

3 Инициаторы и результанты назначаются в соответствии с определением асинхронного процесса.

Определение 3.9. Модельную и предметную интерпретации асинхронного процесса, удовлетворяющие условиям 1-3, называют диаграммой переходов.

Определение 3.10. Ситуацию s_i диаграммы переходов называют конфликтной, если:

1) в ситуации s_i существует компонента z_k, значение которой помечено звездочкой;

2) существует ситуация s_j такая, что s_iFs_j, значения компоненты z_k в s_i и s_j совпадают и в ситуации s_j значение компоненты z_k звездочкой не помечено.

Определение 3.11. Диаграмму переходов называют полумодулярной, если она не содержит конфликтных ситуаций.

Пример 3.7. Диаграмма переходов, соответствующая некоторому автономному АП, представленная на рисунке 3.4, не содержит конфликтных переходов и поэтому является полумодулярной.

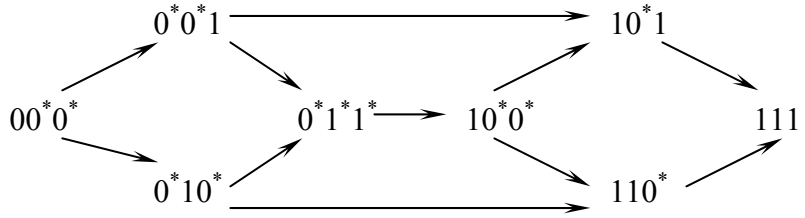


Рисунок 3.4 – Полумодулярная диаграмма переходов

Определение 3.12. Диаграмма переходов называется управляемой, если она соответствует управляемому асинхронному процессу.

Определение 3.13. Моделью Маллера асинхронного процесса называется система булевых уравнений:

$$z_i = f_i(z_1, ..., z_i, ..., z_n), \qquad (3.1)$$

где $i = 1, 2, ..., n$.

В некоторых случаях модель Маллера дополняют указанием исходных значений переменных $z_1, ..., z_n$. Если модель соответствует неавтономному АП, то эти исходные значения соответствуют инициаторам АП.

Установим соответствие между диаграммой переходов и системой уравнений. Для нахождения уравнений можно воспользоваться таблицей истинности, составленной в соответствии со следующими правилами:

1) в левой части таблицы выписываются все возможные наборы переменных диаграммы, возбужденные переменные помечаются звездочками;

2) в правой части таблицы против каждого набора из левой части записывается набор, в котором все переменные, не помеченные звездочками, сохраняют свои значения, а помеченные принимают противоположные значения;

3) не встречающиеся в диаграмме переходов наборы в правой части таблицы могут быть доопределены произвольно.

Пример 3.8. По приведенной на рисунке 3.4 диаграмме переходов построим модель Маллера. Составим таблицу истинности (таблица 3.1).

Таблица 3.1 – Таблица истинности

z_1	z_2	z_3	f_1	f_2	f_3	z_1	z_2	z_3	f_1	f_2	f_3
0	0^*	0^*	0	1	1	1	0^*	0^*	1	1	1
0^*	0^*	1	1	1	1	1	0^*	1	1	1	1
0^*	1	0^*	1	1	1	1	1	0^*	1	1	1
0^*	1^*	1^*	1	0	0	1	1	1	1	1	1

Систему уравнений можно получить по таблице путем применения любой из известных процедур минимизации булевых функций /10/. Модель Маллера будет иметь вид:

$$\begin{cases} z_1 = z_1 \vee z_2 \vee z_3; \\ z_2 = z_1 \vee \overline{z_2} \vee \overline{z_3}; \\ z_3 = z_1 \vee \overline{z_2} \vee \overline{z_3}. \end{cases}$$

Модель Маллера позволяет описывать АП подробно, с точностью до собственных функций элементов соответствующей схемы.

Определение 3.14. В модели Маллера вектор $\alpha = \alpha_1, \ldots, \alpha_n, \alpha_i \in \{0, 1\}$ значений всех переменных $z_1, ..., z_n$ называется состоянием схемы. Говорят, что переменная z_i возбуждена в состоянии α, если $\alpha_i \neq f_i(\alpha)$ и устойчива в противном случае.

Каждая переменная модели Маллера соответствует выходу элемента моделируемой схемы и элемент работает следующим образом. Будучи возбужден, элемент через некоторое время в результате срабатывания переходит в устойчивое состояние, изменив значение выхода. Если же в возбужденном состоянии набор значений на его входах изменится так, что условия возбуждения будут сняты, то элемент оказывается в устойчивом состоянии, не изменяя значения выхода (не срабатывая).

Из состояния α модель может перейти во всякое состояние β, которое отличается от α только значениями каких-либо переменных, возбужденных в α. Говорят, что α непосредственно предшествует состоянию β, а β непосредственно следует за α (непосредственно достижимо из α). Отношение непосредственного следования в модели Маллера обозначается через $\alpha \to \beta$. Если $\alpha \to \beta$ и α отличается от β только значением переменной z_i (α и β соседние по z_i), то это обозначается $\alpha \overset{1}{\underset{z_i}{\to}} \beta$ или $\alpha \overset{1}{\to} \beta$.

3.4.3 Сигнальные графы

Определение 3.15. Маркированный граф – это ориентированный граф, дуги которого маркируются точками.

Точка на всех дугах, входящих в некоторую вершину маркированного графа, означает ее возбуждение. Через непредсказуемый, но конечный промежуток времени возбужденная вершина срабатывает. При этом появляются точки на всех ее выходных дугах и снимается по одной точке со всех входных.

Все вершины, в которые входит более одной дуги, называются синхронизаторами, а вершины, из которых выходит более одной дуги - бифуркаторами.

Динамика функционирования маркированного графа описывается сменой его маркировок, порождаемых описанным выше правилом и заданной исходной маркировкой.

Определение 3.16. Маркированный граф называется живым, если каждая его вершина либо возбуждена, либо может быть возбуждена посредством некоторой последовательности срабатываний возбужденных вершин.

Определение 3.17. Маркированный граф называется безопасным относительно заданной маркировки если:

1) его маркировка живая;

2) ни одна дуга не содержит более одной точки;

3) нет последовательности возбуждений, приводящей к появлению двух или более точек хотя бы на одной дуге.

Маркированный граф является одной из модельных интерпретацией АП. Ситуация в графе - это текущая маркировка, отношение непосредственного следования ситуаций F задается правилом смены маркировки, инициаторы и результанты назначаются в соответствии с определением АП.

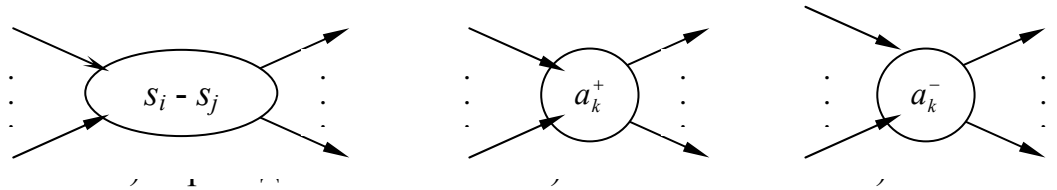
Маркированные графы могут моделировать параллельность (есть синхронизация), но не могут моделировать конфликты (нет вершин с альтернативными выходами).

Предметная интерпретация маркированных графов может быть введена, следующим образом.

1 Вершины графа типа, изображенного на рисунке 3.5 *а*, называют переходами. При возбуждении вершин этого типа (наличии точек на всех входных дугах) инициируется переход s_i - s_j.

2 При достижении s_j точки со всех входных дуг снимаются и устанавливаются точки на всех выходных дугах. Любая промежуточная ситуация перехода не вызывает изменения маркировки.

3 Ситуации переходов можно структурировать, поставив им в соответствие некоторые двоичные наборы. При возбуждении вершины типа, изображенного на рисунке 3.5 б (3.5 в), осуществляется изменение значения некоторой компоненты ситуации из 0 в 1 (из 1 в 0).



а) переход б) из 0 в 1 в) из 1 в 0

Рисунок 3.5 - Три типа вершин в сигнальном графе

Определение 3.18. Сигнальным графом называется предметная интерпретация маркированного графа, удовлетворяющая условиям 1-3.

Таким образом, сигнальный граф позволяет сократить описание некоторого класса АП: полное описание перехода можно заменить единственным переходом от инициатора к результанту.

Пример 3.9. Сигнальный граф, соответствующий ранее рассмотренной диаграмме переходов (рисунок 3.4), изображен на рисунке 3.6.

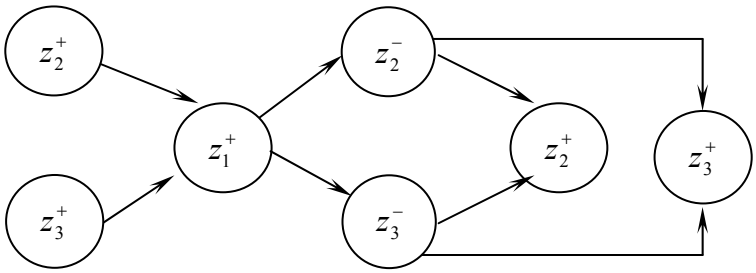


Рисунок 3.6 – Сигнальный граф

Вопросы и задания для самоконтроля

1 Приведите примеры дискретных систем в ЭВМ.

2 Перечислите четыре основных черты дискретных систем.

3 Дайте формальное определение асинхронного процесса.

4 Назовите существующие частные виды асинхронных процессов.

5 Какой асинхронный процесс называется эффективным, автономным?

6 Приведите пример асинхронного процесса, который является простым, но не управляемым; управляемым, но не простым.

7 В чем суть операции структурирования асинхронного процесса?

8 Назовите этапы интерпретации асинхронного процесса.

9 Приведите примеры модельной и предметной интерпретации асинхронного процесса.

10 Какая интерпретация асинхронного процесса называется диаграммой переходов?

11 Какая ситуация диаграммы переходов является конфликтной?

12 Приведите пример полумодулярной диаграммы переходов.

13 Расставьте на диаграмме переходов, приведенной на рисунке 3.7, возбужденные переменные и постройте по ней модель Маллера.

14 Назначьте инициаторы и результанты для диаграммы на рисунке 3.7 так, чтобы процесс был, если возможно, эффективным, управляемым, простым.

15 Опишите динамику функционирования маркированного графа.

16 Какая интерпретация маркированного графа называется сигнальным графом?

17 Постройте сигнальный граф, соответствующий диаграмме переходов на рисунке 3.7.

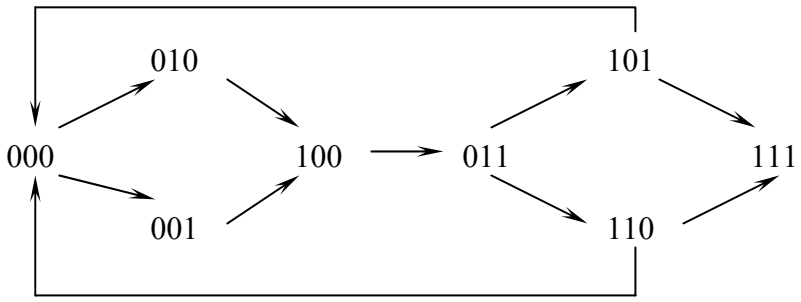


Рисунок 3.7 – Граф асинхронного процесса P_3

Тесты проверки усвоения материала

1 Системы, обладающие конечным числом состояний, называются:

а) асинхронными;

б) фазовыми;

в) последовательными;

г) дискретными;

д) согласованными.

2 Асинхронный процесс, множество инициаторов и результантов которого пусто, называется:

а) дескрептивным;

б) простым;

в) управляемым;

г) эффективным;

д) автономным.

3 Для графа процесса, заданного на рисунке 3.8, справедливо отношение непосредственного следования:

а) $S_2F^3S_7$;

б) $S_2F^3S_5$;

в) $S_2F^3S_8$;
г) $S_1F^3S_7$;
д) $S_1F^3S_4$.

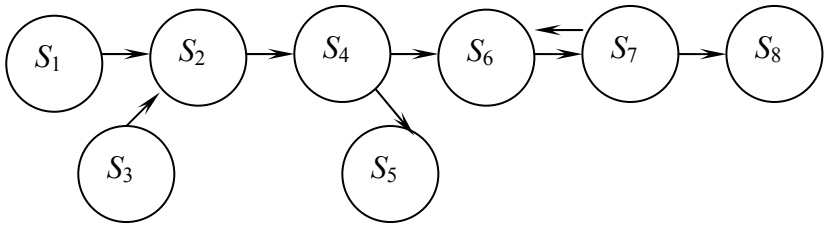


Рисунок 3.8 – Граф процесса P_4

4 Для графа процесса, заданного на рисунке 3.8, справедливо отношение достижимости:
а) S_2MS_3;
б) S_7MS_6;
в) S_2MS_1;
г) S_3MS_6;
д) S_3FS_4.

5 Множество инициаторов I асинхронного процесса должно удовлетворять условию:
а) если iFs_k, $i \in I$, то $s_k \in I$;
б) если s_kFi, $i \in I$, $s_k \notin I$, то из s_kFs_l следует $s_l \notin I$;
в) если iFs_k, $i \in I$, $s_k \notin I$, то из s_kFs_l следует $s_l \notin I$;
г) если iFs_k, $i \in I$, $s_k \notin I$, то из iFs_l следует $s_l \notin I$;
д) если s_lFs_k, $s_k \notin I$, то из iFs_k, $i \in I$ следует $s_l \notin I$.

6 Множество результантов R эффективного асинхронного процесса должно удовлетворять условию:
а) $(\forall s \in S \backslash I)\ (\exists\ r \in R)$: sMr;
б) $(\forall s \in S \backslash R)\ (\exists\ r \in R)$: sMr;
в) $(\forall r \in R)\ (\exists\ s \in S \backslash R)$: rMs;
г) $(\forall r \in R)\ (\exists\ s \in S)$: sMr;
д) $(\forall s \in S)\ (\exists\ r \in R)$: sMr.

7 В графе эффективного асинхронного процесса все циклы должны состоять из:
а) инициаторов;
б) инициаторов и результантов;
в) любых ситуаций, кроме инициаторов;
г) результантов;
д) любых ситуаций, кроме результантов.

8 На графе асинхронного процесса, представленного на рисунке 3.8, в один класс эквивалентности попадают ситуации:

а) S_1 и S_2;

б) S_1 и S_3;

в) S_6 и S_7;

г) S_7 и S_8;

д) S_5 и S_8.

9 Для множества результантов R простого асинхронного процесса должно выполняться условие:

а) $(\forall s \in S \backslash R)\ (\forall r \in R)\ sFr \Rightarrow s \in R$;

б) $(\forall s \in S)\ (\forall r \in R)\ sFr \Rightarrow s \in R$;

в) $(\forall s \in S \backslash R)\ (\forall r \in R)\ sFr \Rightarrow s \notin R$;

г) $(\forall s \in S)\ (\forall r \in R)\ rFs \Rightarrow s \notin R$;

д) $(\forall s \in S)\ (\forall r \in R)\ sFr \Rightarrow s \notin R$.

10 Асинхронный процесс, граф которого представлен на рисунке 3.8, эффективный при:

а) $I = \{S_1, S_3\}$; $R = \{S_6, S_7, S_8\}$;

б) $I = \{S_1, S_3\}$; $R = \{S_5, S_6, S_7, S_8\}$;

в) $I = \{S_1, S_3\}$; $R = \{S_5, S_8\}$;

г) $I = \{S_1, S_3\}$; $R = \{S_5, S_7, S_8\}$;

д) $I = \{S_1\}$; $R = \{S_5, \mathrm{S}_6, S_7, S_8\}$.

11 Вычислительный процесс, граф которого представлен на рисунке 3.8, при $I = \{S_1, S_3\}$ и $R = \{S_6, S_7, S_8\}$ является:

а) асинхронным;

б) эффективным;

в) управляемым;

г) простым;

д) автономным.

12 Для эффективного асинхронного процесса любой класс эквивалентности ситуаций, не принадлежащих к результантам, состоит из:

а) инициаторов;

б) одной ситуации;

в) двух ситуаций;

г) множества ситуаций, кроме инициаторов и результантов;

д) не менее двух ситуаций.

13 Протоколу простого асинхронного процесса, заданного графом на рисунке 3.9, при $I = \{S_1\}$ и $R = \{S_5\}$ принадлежит пара:

а) $\{S_1; S_5\}$;
б) $\{S_1; S_4\}$;
в) $\{S_1; S_3\}$;
г) $\{S_5; S_1\}$;
д) $\{S_5; S_2\}$.

14 Отношение $Q \subset I \times R$, где I – множество инициаторов асинхронного процесса, R – множество его результантов, называется:
а) предметной интерпретацией асинхронного процесса;
б) модельной интерпретацией асинхронного процесса;
в) протоколом асинхронного процесса;
г) структурированием асинхронного процесса;
д) траекторией асинхронного процесса.

15 В диаграмме переходов, представленной на рисунке 3.9, конфликтной является ситуация:
а) S_2;
б) S_4;
в) S_3;
г) S_5;
д) S_1.

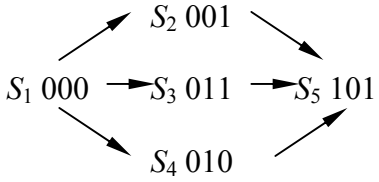


Рисунок 3.9 – Диаграмма переходов процесса P_5

16 В диаграмме переходов, представленной на рисунке 3.9, все компоненты возбуждены в ситуации:
а) S_2;
б) S_4;
в) S_3;
г) S_5;
д) S_1.

17 Переходу $S_1 - S_4$ диаграммы на рисунке 3.9 в маркированном графе соответствует вершина:
а) z_2^-;
б) z_3^+;
в) z_3^-;
г) z_2^+;
д) z_1^+.

18 Диаграмма переходов асинхронного процесса, не содержащая конфликтных ситуаций, называется:

а) полумодулярной;

б) управляемой;

в) автономной;

г) эффективной;

д) простой.

19 Маркированный граф, каждая вершина которого возбуждена или может быть возбуждена в ходе срабатывания возбужденных вершин, называется:

а) безопасным;

б) живым;

в) полумодулярным;

г) устойчивым;

д) эффективным.

20 Вершина маркированного графа, представленная на рисунке 3.10, соответствует:

а) инициации перехода;

б) изменению значения компоненты из 0 в 1;

в) изменению значения компоненты из 1 в 0;

г) промежуточной ситуации;

д) конфликтной ситуации.

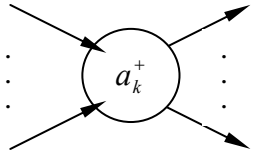


Рисунок 3.10 – Вершина маркированного графа

4 Взаимодействие процессов

4.1 Классификация вычислительных процессов

Определение 4.1. Параллельными называются такие последовательные вычислительные процессы, которые одновременно находятся в каком-либо активном состоянии /11/.

Т.е. параллельными будем считать не только процессы, одновременно развивающиеся на различных процессорах, каналах и устройствах ввода/вывода, но и те последовательные процессы, которые разделяют центральный процессор и хотя бы частично перекрываются во времени.

Два параллельных процесса могут быть независимыми (*independing processes*) либо взаимодействующими (*cooperating processes*).

Определение 4.2. Независимыми называются процессы, множества переменных которых не пересекаются.

Под переменными в этом случае понимают файлы данных и области оперативной памяти, сопоставленные определенным в программе и промежуточным переменным.

Определение 4.3. Процессы, совместно использующие некоторые (общие) переменные, такие, что выполнение одного процесса может повлиять на выполнение другого, называются взаимодействующими.

Определение 4.4. Взаимодействующие процессы, которые действуют относительно независимо, но имеют доступ к общим переменным, называются конкурирующими.

Определение 4.5. Взаимодействующие процессы, выполняющие общую совместную работу таким образом, что результаты вычислений одного процесса в явном виде передаются другому, то есть их работа построена именно на обмене данными, называются сотрудничающими.

Определение 4.6. Ресурсы, которые не допускают одновременного использования несколькими процессами, называются критическими.

Пример 4.1. Рассмотрим работу двух процессов P_1 и P_2 с общей переменной X. Пусть оба процесса асинхронно независимо один от другого изменяют значение переменной X (например, увеличивают на единицу), считывая ее значение в локальную область памяти R_i (имя переменной для процесса с номером i). Каждый процесс выполняет некоторые последовательности операций во времени, из которых мы рассмотрим только операторы, работающие с общей переменной X. Операторам процесса P_1 присвоим номера от 1 до 3, а процесса P_2 – от 4 до 6.

Поскольку процессы могут иметь различные скорости исполнения, то может быть любая последовательность выполнения операций во времени. Если сначала будут выполнены все операции процесса P_1, а потом - все операции процесса P_2, то в итоге переменная X получит значение, равное X+2 (рисунок 4.1).

P_1:	(1)$R_1:=X$	(2)$R_1:=R_1+1$	(3)$X:=R_1$			
P_2:				(4)$R_2:=X$	(5)$R_2:=R_2+1$	(6)$X:=R_2$

t

4.1 - Первый вариант развития событий при выполнении процессов

Если в промежуток времени между выполнением операций 1 и 3 будет выполнена хотя бы одна из операций 4-6 (рисунок 4.2), то значение переменной X после выполнения всех операций будет равно X+1.

P_1:	(1)$R_1:=X$		(2)$R_1:=R_1+1$		(3)$X:=R_1$	
P_2:		(4)$R_2:=X$		(5)$R_2:=R_2+1$		(6)$X:=R_2$

t

4.2 - Второй вариант развития событий при выполнении процессов

Если бы процессы P_1 и P_2 осуществляли продажу билетов в кассе и переменная X фиксировала количество уже проданных, то в результате некоррект-

ного взаимодействия было бы продано несколько билетов на одно и то же место.

Подобное некорректное исполнение конкурирующих процессов происходит вследствие нерегламентированного доступа к разделяемым переменным.

Определение 4.7. Участки программ, в которых происходит обращение к критическим ресурсам, называются критическими секциями или критическими интервалами (*Critical Section - CS*).

4.2 Классические задачи взаимодействия асинхронных процессов

4.2.1 Задача взаимного исключения

Пусть имеются два (или более) циклических процесса, в которых есть абстрактные критические секции, то есть каждый из процессов состоит из двух частей: некоторого критического интервала и оставшейся части кода, в которой нет работы с общими (критическими) переменными. Эти два процесса асинхронно разделяют во времени единственный процессор либо выполняются на отдельных процессорах, каждый из которых имеет доступ к некоторой общей области памяти, с которой работают критические секции (рисунок 4.3).

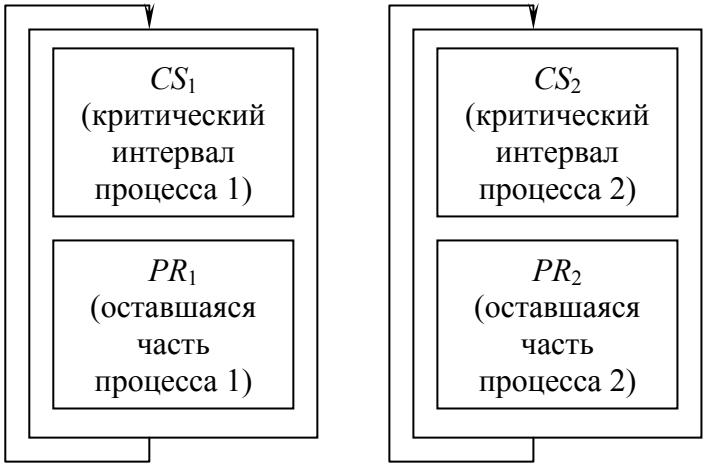


Рисунок 4.3 – Модель взаимодействующих процессов

Необходимо согласовать работу параллельных процессов по использовании критического ресурса таким образом, чтобы выполнялись следующие требования:

- в любой момент времени только один процесс должен находиться в своей критической секции;
- ни один процесс не должен находиться в своей критической секции бесконечно долго;
- ни один процесс не должен ждать бесконечно долго входа в свой критический интервал;

- никакой процесс, бесконечно долго находящийся вне своей критической секции, не должен задерживать выполнение других процессов, ожидающих входа в свои критические секции;

- если два процесса хотят войти в свои критические интервалы, то принятие решения о том, кто первым войдет в критическую секцию, не должно откладываться бесконечно долго;

- если процесс, находящийся в своем критическом интервале, завершается либо естественным, либо аварийным путем, то режим взаимоисключения должен быть отменен, с тем чтобы другие процессы получили возможность входить в свои критические секции.

4.2.2 Задача «производитель-потребитель»

Имеется два типа взаимодействующих процессов с жестко распределенными между ними функциями. Процессы-производители вырабатывают сообщения, предназначенные для восприятия и обработки другими процессами-потребителями. Процессы взаимодействуют через некоторую обобщенную область памяти, которая по смыслу является критическим ресурсом. В эту область процесс-производитель должен поместить очередное сообщение, а процесс-потребитель - считывать очередное сообщение. Необходимо согласовать работу этих процессов по обмену сообщениями таким образом, чтобы выполнялись следующие требования:

- требования задачи взаимного исключения по отношению к критическому ресурсу - обобщенной памяти для хранения сообщения;

- учитывается состояние обобщенной области памяти, характеризующей возможность или невозможность посылки (принятия) очередного сообщения.

Попытка процесса-производителя поместить очередное сообщение в область, из которой не было считано предыдущее сообщение процессом-потребителем, должна быть блокирована. Процесс-потребитель должен быть либо оповещен о невозможности помещения сообщения, либо переведен в состояние ожидания возможности поместить очередное сообщение через некоторое время в область памяти, по мере ее освобождения. Аналогично должна быть заблокирована попытка процесса-потребителя считать сообщение из области в ситуации, когда процесс-производитель не поместил туда очередного сообщения. В этом случае необходимо выдать сообщение о невозможности считывания либо перевести процесс-потребитель в состояние ожидания поступления очередного сообщения. Если используется вариант с ожиданием изменения состояния обобщенной области для хранения сообщений, необходимо обеспечить перевод ожидающих процессов в состояние готовности всякий раз, когда изменится состояние области.

Множественность постановки задачи «производитель-потребитель» определяется тем, что число процессов-потребителей и процессов-производителей может быть больше одного. Каждый из таких процессов может устанавливать не только одностороннюю, но и двухстороннюю связь через од-

ну и ту же обобщенную область или другие области. Области могут хранить большое количество сообщений.

4.2.3 Задача «читатели-писатели»

В отношении некоторой области памяти, являющейся по смыслу критическим ресурсом для параллельных процессов, работающих с ней, выделяется два типа процессов. Первый тип - процессы-читатели, которые одновременно считывают информацию из области, если это допускается при работе с конкретным устройством памяти. Второй тип - процессы-писатели, которые записывают информацию в область и могут делать это, только исключая друг друга и процессы-читатели, т.е. запись должна выполняться на основании решения задачи взаимного исключения.

Имеются различные варианты взаимодействия между процессами-писателями и процессами-читателями. Наиболее широко распространены следующие.

1 Устанавливается приоритетность в использовании критического ресурса процессам-читателям. Если хотя бы один процесс-читатель пользуется ресурсом, то он закрыт для использования всем процессам-писаиелям.

2 Устанавливается приоритетность в использовании критического ресурса процессами-писателями. При появлении запроса от процесса-писателя необходимо закрыть ресурс для использования всем процессам-читателям, которые выдадут запрос позже него.

Наиболее характерная область использования этой задачи - построение файловых систем операционной системы.

4.2.4 Задача «обедающие философы»

Данная задача имеет место при построении систем распределения ресурсов в составе операционной системы. В рамках этой задачи формулируются требования на синхронизацию работы процессов, которые совместно используют пересекающиеся группы ресурсов.

Для примера рассмотрим случай с тремя процессами и тремя ресурсами. Пусть имеются три параллельных процесса X, Y и Z и три ресурса R_1, R_2 и R_3. Особенность развития процессов такова, что для пребывания процесса X в активном состоянии ему требуется выделить одновременно ресурсы R_1 и R_2, для пребывания процесса Y - ресурсы R_2 и R_3, для пребывания процесса Z - ресурсы R_3 и R_1. Скорости развития процессов произвольны. Переходы из активного в другие состояния происходят в непредсказуемые моменты.

Необходимо обеспечить максимальное параллельное и правильное развитие процессов. Синхронизация в данном случае заключается в определенном упорядочении действий процессов по захвату ресурсов во избежание возможных блокировок одними процессами других /12/.

4.3 Проблема тупиков

Определение 4.8. Ситуация, возникающая при параллельном исполнении процессов, при которой два или более процесса все время находятся в заблокированном состоянии, называется дедлоком (*Dead lock* - смертельное объятие) тупиком или клинчем.

В мультипрограммной системе процесс находится в состоянии тупика, если он ждет события, которое никогда не произойдет. Тупики чаще всего возникают из-за конкуренции несвязанных параллельных процессов за ресурсы вычислительной системы, но иногда к тупикам приводят и ошибки программирования.

Ресурсы системы можно разделить на два класса - повторно используемые (или системные) ресурсы (типа *RR* или *SR* - *reusable resource* или *system resource*) и потребляемые (или расходуемые) ресурсы (типа *CR* - *consumable resource*).

Определение 4.9. Повторно используемый ресурс (*SR*) есть конечное множество идентичных единиц со следующими свойствами:

- число единиц ресурса постоянно;
- каждая единица ресурса или доступна, или распределена одному и только одному процессу;
- никакой процесс не может оказывать какое-либо влияние ни на один ресурс, если он ему не принадлежит.

К системным ресурсам относят такие компоненты аппаратуры, как основная память, внешняя память, периферийные устройства, процессоры, программное и информационное обеспечение, такое как файлы данных, таблицы и «разрешение войти в критическую секцию».

Определение 4.10. Расходуемый ресурс (*CR*) есть ресурс, обладающий следующими особенностями:

- число доступных единиц некоторого ресурса типа *CR* изменяется по мере того, как приобретаются (расходуются) и освобождаются (производятся) отдельные их элементы выполняющимися процессами, и такое число единиц ресурса является потенциально неограниченным; процесс «производитель» увеличивает число единиц ресурса, освобождая одну или более единиц, которые он «создал»;
- процесс «потребитель» уменьшает число единиц ресурса, сначала запрашивая и затем приобретая (потребляя) одну или более единиц. Единицы ресурса, которые приобретены, в общем случае не возвращаются ресурсу, а потребляются (расходуются).

Потребляемыми ресурсами являются прерывания от таймера и устройств ввода/вывода; сигналы синхронизации процессов; сообщения, содержащие запросы на различные виды обслуживания или данные, а также соответствующие ответы.

Пример 4.2. Тупик на ресурсах типа *CR* и *SR*.

Пусть некоторый процесс P_1 должен обменяться сообщениями с процессом P_2 и каждый из них запрашивает некоторый ресурс R, причем P_1 требует

три единицы этого ресурса для своей работы, а P_2 - две единицы и только на время обработки сообщения. Всего имеются только четыре единицы ресурса R. Запрос ресурса реализуется через процедуру *REQUEST*(R, N) - запрос N единиц ресурса R, а освобождение – через процедуру *RELEASE*(R, N) - возврат N единиц ресурса R. Обмен сообщениями осуществляется через почтовый ящик MB. Фрагменты программ P_1 и P_2 имеют вид:

```
    …
P1: REQUEST(R, 3);
    …
    SEND_MESSAGE(P2, сообщение, MB);{отправить сообщение P2 в MB}
    WAIT_ANSWER(ответ, MB); {ждать ответ в MB }
    …
    RELEASE(R, 3);
    …
    …
P2: WAIT_MESSAGE(P1, сообщение, MB);{ждать сообщение от P1 в MB}
    REQUEST(R, 2);
    ОБРАБОТКА_СООБЩЕНИЯ;
    RELEASE(R, 2);
    SEND_ANSWER(ответ, MB); {послать ответ в MB }
    …
```

Данные процессы всегда будут попадать в тупик. Процесс P_2, если будет выполняться первым, сначала ожидает сообщения от процесса P_1, после чего будет заблокирован при запросе ресурса R, часть которого уже отдана P_1. Процесс P_1, завладев частью ресурса R, будет заблокирован на ожидании ответа от P_2, которого никогда не получит, так как для этого P_2 нужно получить ресурс R, находящийся в распоряжении P_1. Тупика можно избежать при условии, что на время ожидания ответа от P_2 процесс P_1 будет отдавать хотя бы одну единицу ресурса R, которыми он владеет. В данном примере причиной тупика являются ошибки программирования.

В общем случае для возникновения тупика необходимо одновременное выполнение четырех условий:

- взаимного исключения, при котором процессы осуществляют монопольный доступ к ресурсам;

- ожидания, при котором процесс, запросивший ресурс, ждет до тех пор, пока запрос не будет удовлетворен, при этом удерживая ранее полученные ресурсы;

- отсутствия перераспределения, при котором ресурсы нельзя отобрать у процесса, если они ему уже выделены;

- кругового ожидания, при котором существует замкнутая цепь процессов, каждый из которых ждет ресурс, удерживаемый его предшественником в этой цепи.

4.4 Средства синхронизации взаимодействующих вычислительных процессов

4.4.1 Блокировка памяти

Данное средство запрещает одновременное исполнение двух (и более) команд, которые обращаются к одной и той же ячейке памяти, хранящей значение разделяемой переменной.

Рассмотрим возможный вариант решения задачи взаимного исключения, использующий только блокировку памяти, предложенный математиком Деккером.

Для описания k параллельно выполняющихся последовательных процессов будем использовать конструкцию типа:

parbegin $S_{11}; S_{12}; ...; S_{1n1}$
and $S_{21}; S_{22}; ...; S_{2n2}$
…
and $S_{k1}; S_{k2}; ...; S_{knk}$
parend;

Конструкция из операторов $S_{11}; S_{12}; ...; S_{1n}$ означает последовательное выполнение оператора за оператором. Конструкция типа *while true do begin* $S_{11}; S_{12}; ...; S_{1n}$ *end* показывает, что каждый процесс может выполняться неопределенное время. Конструкция *begin end* заменяет «пустой» оператор.

Алгоритм Деккера

Алгоритм Деккера основан на использовании трех переменных ПЕРЕКЛ1, ПЕРЕКЛ2 и ОЧЕРЕДЬ. Переменная ПЕРЕКЛ1=*true* тогда, когда процесс P_1 хочет войти в свой критический интервал (аналогично для P_2), а значение переменной ОЧЕРЕДЬ указывает, чье сейчас право сделать попытку входа, при условии, что оба процесса хотят выполнить свои критические интервалы.

Если ПЕРЕКЛ2=*true* и ПЕРЕКЛ1=*false*, то выполняется критический интервал процесса P_2 независимо от значения переменной ОЧЕРЕДЬ. Аналогично для случая ПЕРЕКЛ2=*false* и ПЕРЕКЛ1=*true*. Если же оба процесса хотят выполнить свои критические интервалы, то выполняется критический интервал того процесса, на который указывало значение переменной ОЧЕРЕДЬ. Листинг алгоритма будет иметь вид.

```
var ПЕРЕКЛ1, ПЕРЕКЛ2: boolean;
    ОЧЕРЕДЬ: byte;
begin
    ПЕРЕКЛ1:=false;
    ПЕРЕКЛ2:=false;
    ОЧЕРЕДЬ:=1;
    parbegin
     {P1}while true do
          begin ПЕРЕКЛ1:=true;
            if (ПЕРЕКЛ2=true) and (ОЧЕРЕДЬ=2)
```

```
            then while ОЧЕРЕДЬ=2 do begin end;
            CS1; {Критический интервал P1}
            ОЧЕРЕДЬ:=2; ПЕРЕКЛ1:=false;
            PR1; {P1 после критического интервала}
          end
    and
     {P2}while true do
          begin ПЕРЕКЛ2:=true;
            if (ПЕРЕКЛ1=true) and (ОЧЕРЕДЬ=1)
            then while ОЧЕРЕДЬ=1 do begin end
            CS2; {Критический интервал P2}
            ОЧЕРЕДЬ:=1; ПЕРЕКЛ2:=false;
            PR2; {P2 после критического интервала}
          end
    parend
end.
```

Реализация критических интервалов на основе описанного алгоритма практически не используется из-за сложности его обобщения для *N* процессов.

4.4.2 Операции типа «проверка и установка»

Операция «проверка и установка» является аппаратным средством решения задачи критического интервала. В IBM 360(370) данная операция называлась *TS*(*test and set*). Команда *TS* двухадресная (двухоперандная). Ее действие заключается в том, что процессор присваивает значение второго операнда первому, после чего второму операнду присваивается значение, равное единице. Эта команда является неделимой операцией, то есть между ее началом и концом не могут выполняться никакие другие команды.

В микропроцессорах i80386 и старше есть специальные команды, являющиеся вариантами реализации команды типа «проверка и установка». Это команды *BTC*, *BTS*, *BTK*.

Пример 4.3. Рассмотрим возможный вариант решения проблемы критического интервала с помощью команды *TS*. Введем переменную *common*, которая будет общей для всех процессов, использующих некоторый критический ресурс. Данная переменная будет принимать единичное значение, если какой-либо из взаимодействующих процессов находится в своем критическом интервале. С каждым процессом связана своя локальная переменная, которая принимает значение, равное единице, если данный процесс хочет войти в свой критический интервал. Операция *TS* будет присваивать значение *common* локальной переменной и устанавливать *common* в единицу. Программа решения проблемы критического интервала на примере двух параллельных процессов будет иметь вид.

```
var common, local1, 1ocal2: integer;
begin
   common:=0;
```

```
parbegin
{P1}while true do
        begin
          local1:=1;
          while local1=l do TS(local1, common);
          CS1; {Критический интервал процесса P1}
          common:=0;
          PR1; {P1 после критического интервала}
        end
and
{P2}while true do
        begin
          local2:=l;
          while local2=l do TS(local2, common);
          CS2; {Критический интервал процесса P2}
          common:=0;
          PR2; {P2 после критического интервала}
        end
parend
end.
```

Основным недостатком алгоритма Деккера и операций «проверка и установка» является наличие активного ожидания процессов, что напрасно расходует процессорное время.

4.4.3 Семафорные механизмы

Понятие семафорных механизмов было введено Дейкстрой.

Определение 4.11. Семафор – это переменная специального типа, доступная параллельным процессам для проведения над ней только двух операций – «закрытия» и «открытия», названных соответственно *P*-операцией (*P* – от голландского *Proberen* – проверить) и *V*-операцией (*V* – от голландского *Verhogen* – увеличить).

Эти операции неделимы и являются примитивами относительно семафора, который указывается в качестве параметра операций.

Обобщенный смысл примитива *P*(*S*) состоит в проверке текущего значения семафора *S*, и если оно не меньше нуля, то осуществляется переход к следующей за примитивом операции. В противном случае процесс снимается на некоторое время с выполнения и переводится в состояние «пассивного ожидания».

Операция *V*(*S*) связана с увеличением значения семафора на единицу и переводом одного или нескольких процессов в состояние готовности к центральному процессору.

На практике используется много различных видов семафорных механизмов, отличающихся начальным значением и диапазоном изменения значений

семафора, логикой действия семафорных операций, количеством семафоров, доступных для обработки при использовании отдельного примитива.

Допустимыми значениями семафора являются только целые числа. Если максимально возможное значение семафора равно 1, то семафор называется двоичным или мьютексом. В противном случае семафоры называют *N*-ичными. Возможны варианты реализации, в которых семафорные переменные не могут быть отрицательными, в других – отрицательные значения могут указывать на длину очереди процессов, ожидающих открытия семафора.

Программа, иллюстрирующая взаимное исключение двух параллельных процессов с помощью семафорных операций будет иметь следующий вид.

```
var S: semafor;
begin
  InitSem(S, 1); {присвоить семафору S начальное значение 1}
  parbegin
  {P1}while true do
        begin
          P(S);
          CS1; {Критический интервал процесса P1}
          V(S);
          PR1
        end
  and
  {P2}while true do
        begin
          P(S);
          CS2; {Критический интервал процесса P2}
          V(S);
          PR2
        end
  parend
end.
```

Возможна различная реализации семафорных примитивов. Например, она может иметь следующий вид.

```
P(S): S:=S-1;
      if S<0 then WAIT(S); {остановить процесс и поместить в очередь
                            ожидания к семафору S}
V(S): if S<0 then RELEASE(S); {поместить один из ожидающих процессов
                               очереди семафора S в очередь готовности}
      S:=S+1
```

Неделимость *P*- и *V*-операций в однопроцессорной вычислительной системе можно обеспечить с помощью простого запрета прерываний. В мульти-

процессорных системах необходимо дополнительное аппаратное взаимное исключение доступа для различных процессоров, например, использование неделимых команд типа «проверка и установка» (*TS*).

Основное достоинство использования семафорных операций – отсутствие состояния активного ожидания.

4.4.4 Мониторы Хоара

Определение 4.12. В параллельном программировании монитор - это пассивный набор разделяемых переменных и повторно входимых процедур доступа к ним, которым процессы пользуются в режиме разделения, причем в каждый момент им может пользоваться только один процесс.

Внутренние данные монитора могут быть либо глобальными (относящимися ко всем процедурам монитора), либо локальными (относящимися только к одной конкретной процедуре). Ко всем этим данным можно обращаться только изнутри монитора.

Процесс, желающий получить доступ к разделяемым переменным, должен обратиться к процедуре *REQUEST* (запрос) монитора. Если запрашиваемый ресурс свободен, то процесс получает ресурс и покидает монитор. Иначе процедура выдает команду ожидания *WAIT*(*free*) с указанием условия ожидания, которая переводит обратившийся процесс в конец очереди ожидания вне монитора.

Для возврата занимаемого ресурса системе процесс обращается к процедуре *RELEASE* монитора. Данная процедура выполняет команду извещения (сигнализации) *SIGNAL*, чтобы один из ожидающих процессов мог получить данный ресурс и покинуть монитор. Если в это время очередь ожидания пуста, то оповещение не вызывает никаких других действий кроме того, что монитор, вносит ресурс в список свободных.

Пример 4.4. Простейший монитор для выделения одного ресурса будет иметь следующий вид.

```
monitor Resourse;
condition free; {условие - свободный}
var busy: boolean; {busy - занят}
procedure REQUEST; {запрос}
begin
    if busy then WAIT(free);
    busy:=true;
    TakeOff; {выдать ресурс}
end;
procedure RELEASE;
begin
    TakeOn; {взять ресурс}
    busy:=false;
    SIGNAL(free);
end;
```

```
begin
    busy:=false;
end.
```

Мониторы обеспечивают по сравнению с семафорами значительное упрощение организации взаимодействующих вычислительных процессов и большую наглядность при незначительной потере в эффективности.

4.4.5 Почтовые ящики

Определение 4.13. Почтовый ящик - это информационная структура, поддерживаемая операционной системой, которая используется для обмена сообщениями между взаимодействующими процессами.

Почтовый ящик состоит из головного элемента, в котором находится информация о данном ящике, и из нескольких буферов (гнезд), в которые помещают сообщения. Размер каждого буфера и их количество обычно задаются при образовании почтового ящика.

Почтовые ящики являются системными объектами, и для пользования таким объектом необходимо получить его у операционной системы, что осуществляется с помощью соответствующих запросов. Если объем передаваемых данных велик, то эффективнее отправлять в почтовый ящик адрес нахождения данных.

Почтовый ящик может быть связан с парой процессов, только с отправителем, только с получателем, или его можно получить из множества почтовых ящиков, которые используют все или несколько процессов.

Правила работы почтового ящика зависят от его сложности. В простейшем случае сообщения передаются только в одном направлении. Процесс P_1 может посылать сообщения до тех пор, пока имеются свободные гнезда. Если все гнезда заполнены, то P_1 может либо ждать, либо выполнять другую работу и попытаться послать сообщение позже. Аналогично процесс P_2 может получать сообщения до тех пор, пока имеются заполненные гнезда. Если сообщений нет, то он может либо ждать сообщений, либо продолжать свою работу.

Простую схему работы почтового ящика можно усложнять в нескольких направлениях и получать двунаправленные и многовходовые почтовые ящики. Двунаправленный почтовый ящик, связанный с парой процессов, позволяет подтверждать прием сообщений. Если используется множество гнезд, то каждое из них хранит либо сообщение, либо подтверждение. Чтобы гарантировать передачу подтверждений, когда все гнезда заняты, подтверждение на сообщение помещается в то же гнездо, которое было использовано для сообщения, и оно уже не используется для другого сообщения до тех пор, пока подтверждение не будет получено.

Из-за того, что некоторые процессы не забрали свои сообщения, связь может быть приостановлена. Если каждое сообщение снабдить пометкой времени, то управляющая программа может периодически уничтожать старые сообщения. Процессы могут быть также остановлены в связи с тем, что другие процессы не смогли послать им сообщения. Если время поступления каждого

остановленного процесса в очередь заблокированных процессов регистрируется, то управляющая программа может периодически посылать им пустые сообщения, чтобы они не ждали слишком долго.

При работе с почтовыми ящиками используются следующие основные операции.

1 *SEND_MESSAGE* (*Получатель*, *Сообщение*, *Буфер*) - переписывает сообщение в буфер, помещает его адрес в переменную *Буфер* и добавляет буфер к очереди *Получатель*.

2 *WAIT_MESSAGE* (*Отправитель*, *Сообщение*, *Буфер*) - блокирует процесс, выдавший операцию до тех пор, пока в его очереди не появится какое-либо сообщение. Когда процесс устанавливается на процессор, он получает имя отправителя, текст сообщения и адрес буфера. Затем буфер удаляется из очереди, и процесс может записать в него ответ отправителю.

3 *SEND_ANSWER* (*Результат*, *Ответ*, *Буфер*) - записывает *Ответ* в тот *Буфер*, из которого было получено сообщение, и добавляет буфер к очереди отправителя. Если отправитель ждет ответ, он деблокируется.

4 *WAIT_ANSWER* (*Результат*, *Ответ*, *Буфер*) - блокирует процесс, выдавший операцию до тех пор, пока в *Буфер* не поступит ответ. После того как ответ поступил, и процесс установлен на процессор, *Ответ* переписывается в память процессу, а буфер освобождается. *Результат* указывает, является ли ответ пустым.

Основные достоинства почтовых ящиков:

- процессу не нужно знать о существовании других процессов до тех пор, пока он не получит сообщения от них;
- два процесса могут обмениваться более чем одним сообщением за один раз;
- операционная система может гарантировать, что никакой процесс не вмешается в «беседу» других процессов;
- очереди буферов позволяют процессу-отправителю продолжать работу, не обращая внимания на получателя.

Основным недостатком почтовых ящиков является их статический характер: количество буферов для передачи сообщений через почтовый ящик фиксировано.

4.4.6 Конвейеры и очереди сообщений

Конвейер (*pipe* - программный канал) является средством, с помощью которого можно производить обмен данными между процессами. Принцип работы конвейера основан на механизме ввода/вывода, который используется для работы с файлами в UNIX, то есть задача, передающая информацию, действует так, как будто она записывает данные в файл, а задача, для которой предназначается эта информация, читает ее из этого файла. Операции записи и чтения осуществляются не записями, как это делается в обычных файлах, а потоком байтов, как это было принято в UNIX-системах.

Конвейеры представляют собой буферную память, работающую по принципу *FIFO*, то есть по принципу обычной очереди. Максимальный размер конвейера - 64 Кбайт, т.к. в 16-разрядных мини-ЭВМ, для которых создавалась эта система, нельзя было создать массив данных большего размера.

Функционирование конвейера представляет собой реализацию очереди на массивах. Имеются указатели начала (*head*) и конца очереди (*tail*), которые циклически перемещаются по массиву. В начальный момент оба указателя равны нулю. Добавление самого первого элемента в пустую очередь приводит к тому, что указатели *head* и *tail* принимают значение, равное 1 (в массиве появляется первый элемент). Добавление нового элемента изменяет значение второго указателя. Чтение (и удаление) элемента (читается и удаляется всегда первый элемент очереди) приводит к модификации значения указателя *head*. В результате операций записи и чтения элементов в массиве, моделирующем очередь элементов, указатели перемещаются от начала массива к концу. При достижении указателем значения индекса последнего элемента массива значение указателя вновь становится единичным (если при этом не произошло переполнение массива, то есть количество элементов в очереди не стало больше числа элементов в массиве). Таким образом, массив замыкается в кольцо, организуя круговое перемещение указателей *head* и *tail*, которые отслеживают первый и последний элементы в очереди.

Основными системными запросами для работы с конвейером (на примере *API OS*/2) являются: функция создания конвейера (*DosCreatePipe*); функция чтения из конвейера (*DosRead*); функция записи в конвейер (*DosWrite*).

Очереди сообщений (*Queue*) являются более сложным методом связи между взаимодействующими процессами по сравнению с каналами. С помощью очередей можно из одной или нескольких задач независимым образом посылать сообщения некоторой задаче-приемнику. При этом только процесс-приемник может читать и удалять сообщения из очереди, а процессы-клиенты имеют право только помещать в очередь свои сообщения. Очередь работает только в одном направлении. Если необходима двухсторонняя связь, то нужно создать две очереди.

Работа с очередями имеет следующие отличия от работы с конвейерами.

1 Очереди сообщений, в отличие от конвейеров, предоставляют возможность использовать несколько дисциплин обработки сообщений:

- *FIFO* - сообщение, записанное первым, будет первым и прочитано;
- *LIFO* - сообщение, записанное последним, будет прочитано первым;
- приоритетный - сообщения читаются с учетом их приоритетов;
- произвольный доступ, то есть можно читать любое сообщение.

2 При чтении сообщения из конвейера оно удаляется из него, в очереди этого не происходит, и сообщение можно прочитать несколько раз.

3 В очередях хранятся не сами сообщения, а только их адреса в памяти и размер. Эта информация размещается системой в сегменте памяти, доступном для всех задач, общающихся с помощью данной очереди.

При чтении из очереди задача-приемник должна знать идентификатор процесса (*PID* - *process ID*), который передал сообщение; адрес и длину пере-

данного сообщения; ждать или нет, если очередь пуста; приоритет переданного сообщения; номер освобождаемого семафора, когда сообщение передается в очередь.

Управление работой очереди производится с помощью следующих основных функций: создание новой очереди (*CreateQueue*); открытие существующей очереди (*OpenQueue*); чтение и удаление сообщения из очереди (*ReadQueue*); чтение сообщения без его последующего удаления из очереди (*PeekQueue*); добавление сообщения в очередь (*WriteQueue*); завершение использования очереди (*CloseQueue*); удаление из очереди всех сообщений (*PurgeQueue*); определение числа элементов в очереди (*QueryQueue*).

Вопросы и задания для самоконтроля

1 Назовите классы параллельных вычислительных процессов.

2 Приведите примеры сотрудничающих и конкурирующих вычислительных процессов.

3 Сформулируйте классические задачи взаимодействия асинхронных процессов и назовите области их применения.

4 Какая ситуация при параллельном исполнении программы называется дедлоком? Приведите пример.

5 Перечислите условия, необходимые для возникновения тупика в вычислительной системе.

6 Охарактеризуйте классы ресурсов вычислительной системы.

7 Приведите примеры *SR* и *CR* ресурсов в ЭВМ.

8 Перечислите известные аппаратные средства синхронизации взаимодействующих вычислительных процессов.

9 Запишите на псевдокоде алгоритм Деккера.

10 Покажите, как можно решить проблему критического интервала с помощью операции типа «проверка и установка».

11 Назовите основной недостаток методов блокировки памяти и операции типа «проверка и установка».

12 Раскройте сущность семафорных механизмов Дейкстры.

13 Чем могут отличаться различные реализации семафорных механизмов?

14 Проиллюстрируйте взаимное исключение *n* параллельных процессов с помощью семафорных механизмов.

15 Как можно обеспечить неделимость *P*- и *V*-операций в вычислительной системе?

16 Поясните механизм работы мониторов Хоара.

17 Каковы правила работы почтового ящика в зависимости от его сложности?

18 Запишите основные функции, которые используются для работы с почтовыми ящиками.

19 Назовите основные достоинства почтового ящика как средства синхронизации взаимодействующих вычислительных процессов.

20 Что представляет собой функционирование конвейера?
21 Запишите основные системные запросы для работы с конвейером.
22 Назовите отличия работы очереди от конвейера.
23 Запишите основные функции управления работой очереди.

Тесты проверки усвоения материала

1 Последовательные вычислительные процессы, которые одновременно находятся в каком-либо активном состоянии, называются:
а) согласованными;
б) параллельными;
в) асинхронными;
г) сотрудничающими;
д) конкурирующими.

2 Вычислительные процессы, действующие относительно независимо, но имеющие доступ к общим переменным, называются:
а) независимыми;
б) взаимоисключающими;
в) асинхронными;
г) сотрудничающими;
д) конкурирующими.

3 Классическая задача синхронизации работы процессов, совместно использующих пересекающиеся группы ресурсов, называется:
а) «читатели-писатели»;
б) «потребители-производители»;
в) «о взаимном исключении»
г) «обедающие философы»;
д) «о взаимном обмене».

4 К повторно используемым ресурсам вычислительной системы относятся:
а) программное и информационное обеспечение ЭВМ;
б) прерывания от таймера;
в) прерывания устройств ввода/вывода;
г) сигналы синхронизации процессов;
д) сообщения, содержащие запросы на различные виды обслуживания или данные.

5 К потребляемым ресурсам вычислительной системы относятся:
а) периферийные устройства;
б) основная и внешняя память ЭВМ;
в) прерывания устройств ввода/вывода;
г) программное обеспечение;
д) файлы данных и таблицы.

6 Ситуация, при которой два или более параллельных процесса все время заблокированы, называется:

а) критическим интервалом;

б) устойчивой;

в) круговым ожиданием;

г) дедлоком;

д) взаимным исключением.

7 Решение задачи взаимного исключения процессов, предложенное Деккером, в качестве средства синхронизации использует:

а) семафор;

б) блокировку памяти;

в) монитор;

г) очередь;

д) конвейер.

8 Средство синхронизации взаимодействующих вычислительных процессов с активным ожиданием:

а) семафор;

б) монитор;

в) очередь;

г) блокировка памяти;

д) конвейер.

9 Конструкция типа *while true do begin* S_1; S_2; ...; S_k *end* в параллельном программировании определяет:

а) k параллельно выполняющихся процессов;

б) последовательное выполнение оператора за оператором;

в) «пустой» оператор;

г) бесконечный цикл;

д) последовательность процессов, выполняющихся неопределенное время.

10 Двухадресная машинная команда, присваивающая первому операнду значение второго, а затем второму – единицу:

а) *ST*;

б) *TS*;

в) *BTK*;

г) *BTS*;

д) *BTC*.

11 Допустимое значение семафора:

а) -11;

б) 5.8;

в) *a*;

г) *true*;
д) 0.3E-1.

12 Двоичный семафор называется:
а) монитором;
б) биномом;
в) флагом;
г) тестором;
д) мьютексом.

13 При реализации операции открытия семафора *V*(*S*) за командой *if S*<0 *then RELEASE*(*S*) будет следовать команда:
а) *P*(*V*) {закрыть семафор};
б) *S*:=*S*-1;
в) *S*:=*S*+1;
г) *CS* {критическая секция};
д) *PR* {процесс вне критического интервала}.

14 При реализации операции закрытия семафора *P*(*S*) перед командой *if S*<0 *then WAIT*(*S*) будет команда:
а) *P*(*V*) {закрыть семафор};
б) *S*:=*S*-1;
в) *S*:=*S*+1;
г) *CS* {критическая секция};
д) *PR* {процесс вне критического интервала}.

15 При работе с почтовым ящиком *Получатель*, *Сообщение* и *Буфер* указываются в качестве параметров операции:
а) *WAIT_MESSAGE*;
б) *SEND_ANSWER*;
в) *WAIT_ANSWER*;
г) *SEND_MESSAGE*;
д) *CREATE_MESSAGE*.

16 При работе с почтовым ящиком в операции *WAIT_ANSWER* указываются параметры *Результат, Ответ* и:
а) *Буфер*;
б) *Сообщение*;
в) *Отправитель*;
г) *Получатель*;
д) *Время*.

17 Максимальный размер конвейера:
а) 128 Кбайт;
б) 32 Кбайта;

в) 64 Кбайт;
г) 32 Мбайт;
д) 16 Мбайт.

18 Конвейеры используют дисциплину обслуживания сообщений по принципу:
а) *LIFO*;
б) *FIFO*;
в) приоритетности сообщения;
г) произвольного доступа;
д) *LIFO* и *FIFO*.

19 Чтение сообщения без его последующего удаления из очереди реализует команда:
а) *ReadQueue*;
б) *WriteQueue*;
в) *PurgeQueue*;
г) *QueryQueue*;
д) *PeekQueue*.

20 Команда *QueryQueue* выполняет:
а) чтение и удаление сообщения из очереди;
б) чтение сообщения без его последующего удаления из очереди;
в) добавление сообщения в очередь;
г) определение числа элементов в очереди;
д) удаление из очереди всех сообщений.

5 Модели вычислительных процессов

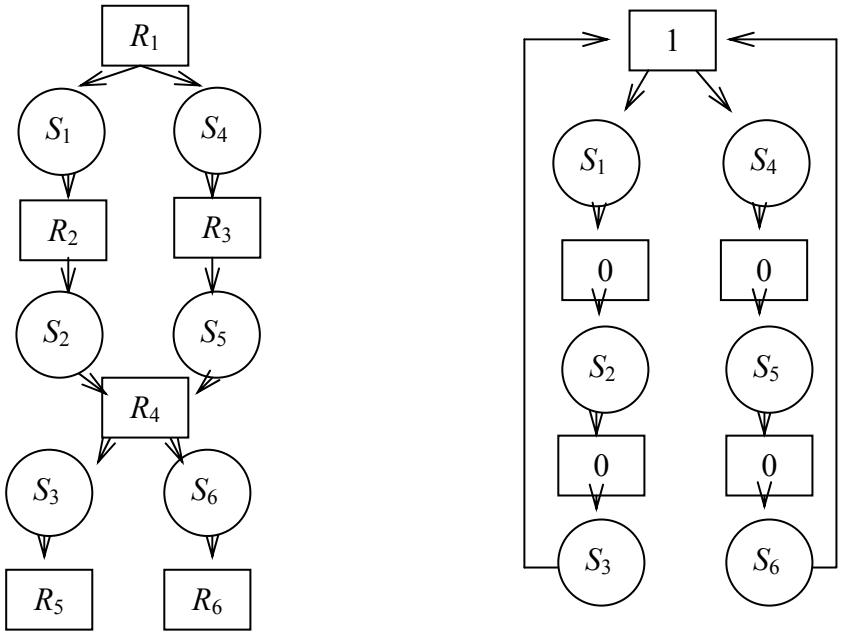
К настоящему времени разработано несколько десятков различных моделей, предназначенных для анализа и моделирования систем с параллельными асинхронными процессами /11, 13/. Рассмотрим четыре из них - вычислительные схемы, модель пространства состояний системы, модель Холта и сети Петри.

5.1 Вычислительные схемы

Определение 5.1. Вычислительная схема - это представление в графической форме асинхронной системы, состоящей из набора операторов (процессов), которые воздействуют на множество «регистров» (данных). Каждая вычислительная схема определяется с помощью двух графов: графа потока данных и графа управления.

Граф потока данных (информационный граф) определяет входные и выходные данные для каждого оператора. Дуга (R_i S_k) от регистра R_i к оператору S_k означает, что данные R_i являются элементом входных данных этого операто-

ра; дуга (S_k R_i) определяет данные R_i как выходные для S_k. Некоторые данные R могут являться выходными для оператора S_i и входными для оператора S_j. Пример графа потока данных для некоторой вычислительной схемы представлен на рисунке 5.1 а; операторы и регистры данных представлены соответственно кружками и прямоугольниками.



а) граф потока данных б) граф управления

Рисунок 5.1 - Пример вычислительной схемы

Граф управления определяет последовательность выполнения операторов. Каждый оператор (представлен кружком) связан с некоторым количеством управляющих счетчиков (представлены прямоугольниками). Каждый из управляющих счетчиков содержит неотрицательное целое число. Текущие значения счетчиков совместно со значениями данных на графе потока данных определяют состояние вычислительной схемы. Пример графа управления представлен на рисунке 5.1 б. Если все счетчики, указывающие на оператор S (то есть входные счетчики), имеют ненулевые значения, то говорят, что оператор S определен. В этом случае он может выполниться, изменив свои выходные регистры в соответствии с графом потока данных и изменив счетчики графа управления по следующему правилу: значения всех входных счетчиков оператора S уменьшаются на единицу, а значения выходных – увеличивается. Причем для каждого выходного счетчика оператора S приращение может быть свое, персональное, и определяется оно с помощью специальной функции от значений регистров.

Определение 5.2. Последовательность операторов S_1, S_2, … , S_n, ..., такая, что каждый оператор S_j определен (то есть его входные счетчики не равны нулю) при тех значениях счетчиков, которые получаются в результате выполнения предшествующих операторов, называется последовательностью исполнения схемы.

Т.к. с операторами не связано никакого особого отсчета времени, то порядок, в котором операторы будут выполняться, не всегда может быть предсказан. Любая допустимая последовательность исполнения является возможной последовательностью событий.

Определение 5.3. Вычислительная схема называется детерминированной, если она вырабатывает одинаковые результаты для всех допустимых последовательностей исполнения.

Пример 5.1. Схема на рисунке 5.1 является детерминированной.

Рассмотрим вычислительную схему на рисунке 5.2. Две последовательности операторов S_1, S_2 и S_3, S_4, как это видно из графа управления, выполняются параллельно и асинхронно. Очевидно, что значения регистров R_2 и R_3 будут различными в зависимости от того, выполняется ли первая последовательность операторов раньше или позже второй. Поскольку граф управления здесь допускает последовательности исполнения, которые приводят к различным результатам, то эта схема недетерминирована.

Определение 5.4. Говорят, что два оператора соперничают в регистре R, если один из них изменяет R, а другой либо изменяет R, либо обращается к нему.

Если два оператора, которые соперничают в некотором регистре, могут быть выполнены в одно и то же время, то говорят, что в схеме существует условие соперничества и такая схема является недетерминированной. Одна из возможных форм недетерминированного исполнения заключается в том, что схема может «зависнуть» (попасть в тупиковую ситуацию).

Вычислительные схемы не являются конструктивной моделью, несмотря на свою интуитивную понятность и возможность сделать вывод о возникновении тупиков в системе.

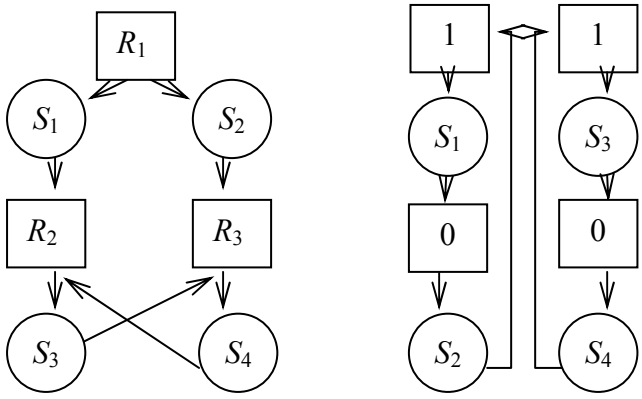


Рисунок 5.2 - Пример недетерминированной вычислительной схемы

5.2 Модель пространства состояний системы

Пусть состояние системы будет сводиться к состоянию различных ресурсов в системе (свободны они или распределены). Состояние системы изменяется процессами, когда они запрашивают, приобретают или освобождают ресурсы; это будут единственно возможные действия, которые принимаются во внимание. Если процесс не блокирован в данном состоянии, он может изменить это

состояние на новое. Но, так как в общем случае невозможно знать заранее, какой путь может избрать произвольный процесс в своей программе (это неразрешимая проблема!), то новое состояние может быть любым из конечного числа возможных. Следовательно, процессы нами будем трактовать как недетерминированные объекты. Введенные ограничения на известные понятия приводят к следующим формальным определениям.

Определение 5.5. Система есть пара $<\sigma, \pi>$, где:

1) σ - множество состояний системы $\{S_1, S_2, ..., S_n\}$;

2) π - множество процессов $\{P_1, P_2, \ldots, P_k\}$.

Определение 5.6. Процесс P_j есть частичная функция, отображающая состояние системы в непустые подмножества ее же состояний. Это обозначается так: P_j: $\sigma \rightarrow \{\sigma\}$.

Если P_j определен на состоянии S, то область значений P_j обозначается как $P_j(S)$. Если $S_k \in P_j(S_l)$, то говорят, что P_j может изменить состояние S_l в состояние S_k посредством операции, и используют обозначение: $S_l \xrightarrow{P_j} S_k$.

Запись $S_l \xrightarrow{*} S_w$ соответствует выполнению одного из трех условий:

1) $S_l = S_w$;

2) $S_l \xrightarrow{P_j} S_w$ для некоторого j;

3) $S_l \xrightarrow{P_j} S_k$ для некоторого j и S_k, и $S_k \xrightarrow{*} S_w$.

Т.е. система может быть переведена посредством $n \geq 0$ операций с помощью $m \geq 0$ различных процессов из состояния S_l в состояние S_w.

Процесс заблокирован в данном состоянии, если он не может изменить состояние системы. Формально это можно определить следующим образом.

Определение 5.7. Процесс P_j заблокирован в состоянии S_l, если не существует ни одного состояния S_k, такого что $S_l \xrightarrow{P_j} S_k$ $(P_j(S_l) = \varnothing)$.

Процесс находится в тупике в данном состоянии S_l, если он заблокирован в состоянии S_l и вне зависимости от того, какие операции произойдут в будущем, процесс остается заблокированным. Формально это можно определить следующим образом.

Определение 5.8. Процесс P_j находится в тупике в состоянии S_l, если для всех состояний S_k, таких что $S_l \xrightarrow{*} S_k$, процесс P_j блокирован в S_k.

Пример 5.2. Определим систему $<\sigma, \pi>$ следующим образом:

1) $\sigma = \{S_1, S_2, S_3, S_4, S_5\}$;

2) $\pi = \{P_1, P_2\}$, где $P_1(S_1) = \{S_2, S_3, S_4\}$; $P_2(S_1) = \{S_2\}$; $P_1(S_2) = \{S_5\}$; $P_2(S_3) = \{S_5\}$; $P_1(S_4) = \{S_5\}$; $P_2(S_4) = \{S_1\}$; $P_1(S_5) = \{S_2\}$.

Примерами возможных последовательностей изменений для этой системы являются $S_1 \xrightarrow{P_1} S_3; S_4 \xrightarrow{P_2} S_1; S_1 \xrightarrow{*} S_5$. Последовательность $S_1 \xrightarrow{*} S_5$ может быть реализована, например, следующим образом: $S_1 \xrightarrow{P_1} S_2; S_2 \xrightarrow{P_1} S_5$ или же $S_1 \xrightarrow{P_1} S_3; S_3 \xrightarrow{P_2} S_5$. Процесс P_2 находится в тупике в состояниях S_2 и S_5; для случая $S_3 \xrightarrow{P_2} S_5$ процесс P_1 не становится заблокированным в S_5.

Диаграмма переходов этой системы изображена на рисунке 5.3.

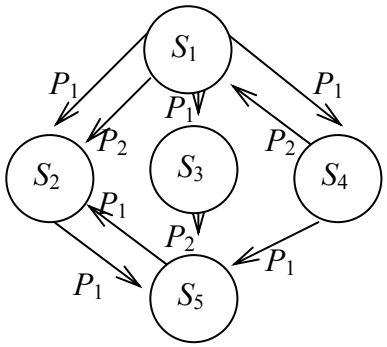


Рисунок 5.3 - Пример системы <σ, π> на 4 состояния

Определение 5.9. Состояние S называется тупиковым, если существует процесс P_i, находящийся в тупике в состоянии S.

Определение 5.10. Состояние S_i безопасное, если для всех состояний S_k, таких что $S_i \xrightarrow{*} S_k$, S_k не является тупиковым состоянием.

Пример 5.3. Граф состояний системы <σ, π> приведен на рисунке 5.4. Здесь состояния S_2, S_3 и S_4 являются безопасными, т.к. из них система никогда не сможет попасть в тупиковое состояние. Состояние S_1 может привести как к безопасным состояниям, так и к опасному состоянию S_5. Состояния S_6 и S_7 являются тупиковыми.

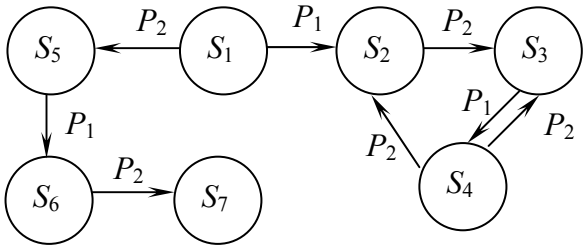


Рисунок 5.4 - Система с безопасными, опасными и тупиковым состояниями

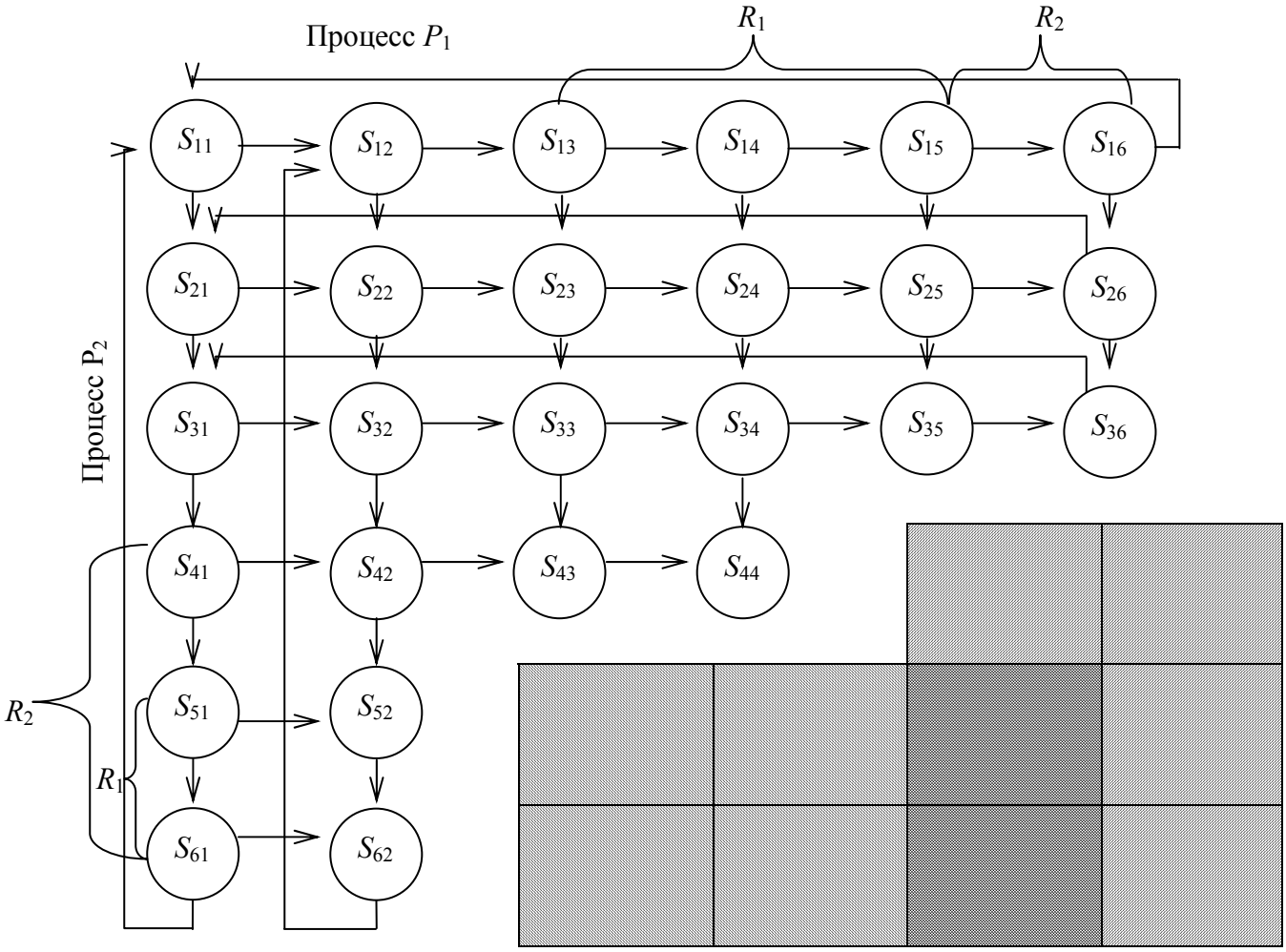
Пример 5.4. Пусть в системе <σ, π> состояния процессов P_1 и P_2, которые используют ресурсы R_1 и R_2, описаны в таблице 5.1.

Таблица 5.1 - Состояния процессов P_1 и P_2

Ресурс/Процесс	Процесс P_1		Процесс P_2	
	R_1	R_2	R_1	R_2
Захватывается в состоянии	S_3	S_5	S_5	S_4
Освобождается в состоянии	S_5	S_6	S_6	S_6

Пусть состояние системы S_{ij} означает, что процесс P_1 находится в состоянии S_j, а процесс P_2 - в состоянии S_i. Возможные изменения в пространстве состояний этой системы изображены на рисунке 5.5. Вертикальными стрелками

показаны возможные переходы из одного состояния в другое для процесса P_2, а горизонтальными - для процесса P_1. Фигурными скобками выделены интервалы удерживания ресурсов процессами. Участки взаимного исключения процессов показаны штриховкой. В данной системе имеется одно опасное состояние S_{43}, попав в него, система неминуемо перейдем в тупиковое состояние S_{44}.



- взаимное исключение по ресурсу R_1;
- взаимное исключение по ресурсу R_2;
- взаимное исключение по ресурсам R_1 и R_2.

Рисунок 5.5 – Пространство состояний системы

5.3 Модель Холта

5.3.1 Описание модели повторно используемых ресурсов

Модель повторно используемых ресурсов Холта была разработана для исследования параллельных процессов и, в частности, проблемы тупиков. Согласно этой модели система представляется как набор (множество) процессов и набор ресурсов, причем каждый из ресурсов состоит из фиксированного числа

единиц. Любой процесс может изменять состояние системы с помощью запроса, получения или освобождения единицы ресурса.

В графической форме процессы и ресурсы представляются квадратами и кружками соответственно. Каждый кружок содержит некоторое количество маркеров (фишек) в соответствии с существующим количеством единиц этого ресурса. Дуга, указывающая из «процесса» на «ресурс», означает запрос одной единицы этого ресурса. Дуга, указывающая из «ресурса» на «процесс», представляет выделение ресурса процессу. Поскольку каждая единица любого ресурса типа *SR* может быть выделена одновременно не более чем одному процессу, то число дуг, исходящих из ресурса к различным процессам, не может превышать общего числа единиц этого ресурса. Такая модель называется графом повторно используемых ресурсов.

Пример 5.5. Одно из состояний системы из трех процессов с ресурсами типа *SR* представлено на рисунке 5.6.

Пусть процесс P_1 запрашивает две единицы ресурса R_1 и одну единицу ресурса R_2. Процессу P_2 принадлежат две единицы ресурса R_1 и одна единица ресурса R_3 и ему нужна одна единица R_2. Процессу P_3 требуется по одной единице ресурсов R_2 и R_3. Предположим, что процесс P_1 получил запрошенную им единицу R_2. Если принято правило, по которому процесс должен получить все запрошенные им ресурсы, прежде чем освободить хотя бы один из них, то удовлетворение запроса P_1 приведет к тупиковой ситуации: P_1 не сможет продолжиться до тех пор, пока P_2 не освободит единицу ресурса R_1, а процесс P_2 не сможет продолжиться до тех пор, пока P_1 не освободит единицу R_2; P_3 будет ждать освобождения R_2 и R_3, которое никогда не произойдет. Причиной этого дедлока являются неупорядоченные попытки процессов войти в критический интервал, связанный с выделением соответствующей единицы ресурса.

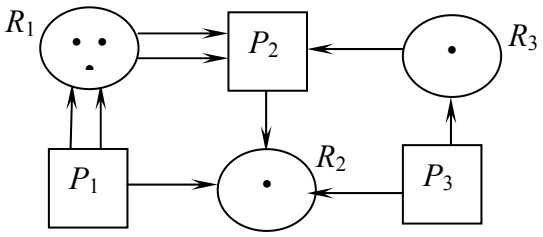


Рисунок 5.6 - Пример модели Холта для системы из трех процессов

5.3.2 Обнаружение тупика посредством редукции графа повторно используемых ресурсов

Сущность процесса редукции графа повторно используемых ресурсов описывается следующими положениями.

1 Процесс P_i, который не является ни заблокированной, ни изолированной вершиной, сокращает граф повторно используемых ресурсов путем удаления всех ребер, входящих в вершину P_i и выходящих из P_i. Эта процедура эквивалентна приобретению процессом P_i всех необходимых ресурсов, а затем ос-

вобождению всех его ресурсов. В результате P_i становится изолированной вершиной графа.

2 Граф повторно используемых ресурсов несокращаем (не редуцируется), если он не может быть сокращен ни одним процессом.

3 Граф ресурсов типа *SR* является полностью сокращаемым, если существует последовательность сокращений, которая удаляет все дуги графа.

Доказана лемма, которая позволяет предложить алгоритмы обнаружения тупика.

Лемма 5.1. Для ресурсов типа *SR* порядок сокращений несуществен; все последовательности ведут к одному и тому же несокращаемому графу.

Теорема 5.1. О тупике. Состояние *S* есть состояние тупика тогда и только тогда, когда граф повторно используемых ресурсов в состоянии *S* не является полностью сокращаемым.

Доказательство.

1 Предположим, что состояние *S* есть состояние тупика и процесс P_i находится в тупике в *S*. Тогда для всех S_j, таких что $S \xrightarrow{*} S_j$, процесс P_i заблокирован в S_i (по определению). Так как сокращения графа идентичны для серии операций процессов, то конечное несокращаемое состояние в последовательности сокращений должно оставить процесс P_i блокированным. Следовательно, граф не является полностью сокращаемым.

2 Предположим, что *S* не является полностью сокращаемым. Тогда существует процесс P_i, который остается заблокированным при всех возможных последовательностях операций редукции в соответствие с леммой. Так как любая последовательность операций редукции графа повторно используемых ресурсов, оканчивающаяся несокращаемым состоянием, гарантирует, что все ресурсы типа *SR*, которые могут когда-либо стать доступными, в действительности освобождены, то процесс P_i навсегда заблокирован и, следовательно, находится в тупике.

Следствие 1. Процесс P_i не находится в тупике тогда и только тогда, когда серия сокращений приводит к состоянию, в котором P_i не заблокирован.

Следствие 2. Если *S* есть состояние тупика (по ресурсам типа *SR*), то, по крайней мере, два процесса находятся в тупике в *S*.

Из теоремы о тупике непосредственно следует и алгоритм обнаружения тупиков. Нужно попытаться сократить граф по возможности эффективным способом; если граф полностью не сокращается, то начальное состояние было состоянием тупика для тех процессов, вершины которых остались в несокращенном графе. Рассмотренная лемма позволяет удобным образом упорядочивать сокращения.

Пусть имеется *n* процессов и *m* ресурсов, тогда граф повторно используемых ресурсов можно представить двумя матрицами размером $n \times m$ и вектором:

1) $D = ||d_{ij}||$ - матрица распределения ресурсов, в которой элемент d_{ij} отражает количество единиц ресурса R_j, распределенного процессу P_i, то есть $d_{ij} = |(R_j,P_i)|$, где (R_j, P_i) - это дуга, ведущая из R_j в P_i, $0 \le i \le n$, $0 \le j \le m$;

2) $N = ||n_{ij}||$, матрица запросов, где $n_{ij} = |(P_i, R_j)|$;

3) $r = (r_1, r_2, ..., r_m)$ - одномерный массив доступных единиц ресурсов, где r_j указывает число доступных (нераспределенных единиц ресурса R_j).

Метод прямого обнаружения тупика заключается в просмотре по порядку матрицы запросов, причем, где возможно, производятся сокращения дуг графа до тех пор, пока нельзя будет сделать ни одного сокращения. Время выполнения такого алгоритма в наихудшем случае пропорционально $m \times n^2$.

Более эффективный алгоритм может быть получен за счет хранения дополнительной информации о запросах. Для каждой вершины процесса P_i определяется счетчик ожиданий w_i, отображающий количество ресурсов (не число единиц ресурса), которые в какое-то время вызывают блокировку процесса. Кроме этого, можно сохранять для каждого ресурса запросы, упорядоченные по размеру (числу единиц ресурса).

Пусть L - текущий список процессов, которые могут выполнить редукцию графа, т.е. $L := \{P_i \mid w_i = 0\}$. Алгоритм выбирает процесс P из списка L, процесс P сокращает граф, увеличивая число доступных единиц r_j всех ресурсов R_j, распределенных процессу P, обновляет счетчик ожидания w_i каждого процесса P_i, который сможет удовлетворить свой запрос на ресурс R_j, и добавляет P_i к L, если счетчик ожидания становится нулевым.

Данный алгоритм имеет максимальное время выполнения, пропорциональное $m \times n$.

5.3.3 Методы обнаружения тупика по наличию замкнутой цепочки запросов

Структура графа обеспечивает простое необходимое (но не достаточное) условие для тупика. Для любого графа $G = <X, E>$ и вершины $x \in X$ пусть $P(x)$ обозначает множество вершин, достижимых из вершины x

Тогда если существует вершина $x \in X$: $x \in P(x)$, то в графе G имеется цикл.

Теорема 5.2. Цикл в графе повторно используемых ресурсов является необходимым условием тупика.

Теорема 5.3. Если S не является состоянием тупика и $S \xrightarrow{P_i} S_T$, то S_T есть состояние тупика в том и только в том случае, когда операция процесса P_i есть запрос и P_i находится в тупике в S_T.

Другими словами, тупик может быть вызван только при запросе, который не удовлетворен немедленно. Значит, проверка на тупиковое состояние может быть выполнена более эффективно, если она проводится непрерывно, то есть по мере развития процессов. Тогда надо применять редукцию графа только после запроса от некоторого процесса P_i и на любой стадии необходимо сначала попытаться его сократить с помощью процесса P_i. Если процесс P_i смог провести сокращение графа, то никакие дальнейшие сокращения не являются необходимыми.

Определение 5.11. Пучок (или узел) в ориентированном графе $G=<X,E>$ - это подмножество вершин $Z \subseteq X$, таких что $\forall x \in Z$, $P(x) = Z$, то есть потомством каждой вершины из Z является само множество Z. Каждая вершина в узле дос-

тижима из каждой другой вершины этого узла, и узел есть максимальное подмножество с этим свойством. В узел дуги должны входить, но не должны выходить.

Определение 5.12. Состояние называется фиксированным, если каждый процесс, выдавший запрос, заблокирован.

Теорема 5.4. Если состояние системы фиксированное, то наличие узла в графе повторно используемых ресурсов является достаточным условием тупика.

Доказательство. Предположим, что граф содержит узел *Z*. Тогда все процессы в этом узле должны быть заблокированы только по ресурсам, принадлежащим *Z*, так как никакие ребра не могут выходить из *Z* по определению. Аналогично, по той же самой причине все распределенные ресурсы узла *Z* принадлежат процессам из *Z*. Наконец, все процессы в *Z* должны быть заблокированы согласно условию фиксированности и определению узла. Следовательно, все процессы в узле *Z* должны быть в тупике.

Теорема 5.5. Граф повторно используемых ресурсов с единичной емкостью ($|R_i| = 1$, (i =1, 2, ..., m)) указывает на состояние тупика тогда и только тогда, когда он содержит цикл.

Доказательство. Необходимость цикла доказывает теорема 5.2. Для доказательства достаточности допустим, что граф содержит цикл, и рассмотрим только процессы и ресурсы, принадлежащие циклу. Так как каждая вершина-процесс должна иметь входящее и исходящее ребро, она должна иметь выданный запрос на некоторый ресурс, принадлежащий циклу, и должна удерживать некоторый ресурс, принадлежащий тому же циклу. Аналогично, каждый ресурс единичной емкости в цикле должен быть захвачен некоторым процессом. Следовательно, каждый процесс в цикле блокируется на ресурсе, который может быть освобожден только некоторым процессом из этого цикла; поэтому процессы в цикле находятся в тупике.

Чтобы обнаружить тупик в случае ресурса единичной емкости, необходимо просто проверить граф повторно используемых ресурсов на наличие циклов.

Алгоритм обнаружения тупика по наличию замкнутой цепочки запросов

Алгоритм был разработан сотрудниками фирмы IBM и использовался в одной из ОС этой компании. Он использует информацию о состоянии системы, содержащуюся в двух таблицах:

1) *RATBL* - таблица текущего распределения (назначения) ресурсов;

2) *PWTBL* - таблица заблокированных процессов (для каждого вида ресурса может быть свой список заблокированных процессов).

При каждом запросе на получение или освобождении ресурсов содержимое этих таблиц модифицируется, а при запросе - анализируется в соответствии со следующим алгоритмом.

1 Запрос от процесса J на занятый ресурс I.

2 Поместить номер ресурса I в *PWTBL* в строке с номером процесса J.

3 Использовать I в качестве смещения в *RATBL*, чтобы найти номер процесса K, который владеет ресурсом.

4 Использовать K в качестве смещения в *PWTBL*.

5 Проверить, ждет ли процесс K освобождения какого-либо ресурса I'. Если нет, то перейти к шагу 6, в противном случае — к шагу 7.

6 Перевести J в состояние ожидания и выйти из алгоритма.

7 Использовать I' в качестве смещения в *RATBL*, чтобы найти номер блокирующего его процесса K'.

8 Проверить $K' = J$. Если нет, то перейти к шагу 9, в противном случае - к шагу 11.

9 Проверить, вся ли таблица *PWTBL* просмотрена. Если да, то перейти к шагу 6, в противном случае - к шагу 10.

10 Присвоить $K := K'$ и перейти к шагу 4.

11 Сделать вывод о наличии тупика с последующим восстановлением.

Конец алгоритма.

Пример 5.6. Рассмотрим следующую последовательность событий.

1 Процесс P_2 занимает ресурс R_1.

2 Процесс P_3 занимает ресурс R_2.

3 Процесс P_3 занимает ресурс R_3.

4 Процесс P_1 занимает ресурс R_4.

Таблица распределения ресурсов (*RATBL*) будет иметь вид таблицы 5.2.

Таблица 5.2 - Таблица распределения ресурсов *RATBL*

Ресурсы	Процессы
1	2
2	3
3	3
4	1

Выполним алгоритм по шагам.

1 Пусть процесс P_1 пытается занять ресурс R_1, тогда $J = 1$, $I = 1$, $K=2$. Процесс K не ждет никакого ресурса I', поэтому процесс P_1 блокируется по ресурсу R_1.

2 Пусть процесс P_2 пытается занять ресурс R_2, тогда $J=2$, $I=2$, $K=3$.

3 Процесс K не ждет никакого ресурса, поэтому процесс P_2 блокируется по ресурсу R_2.

4 Теперь пусть процесс P_3 пытается обратиться к ресурсу R_4. Тогда $J=3$, $I = 4$, $K = 1$, $I' = 1$, $K'= 2$, $K' <> J$, поэтому берем $K = 2$, $I' = 2$, $K' = 3$.

В этом случае $K' = J$, то есть тупик определен. Таблица заблокированных процессов (*PWTBL*) теперь имеет следующий вид таблицы 5.3.

Таблица 5.3 - Таблица заблокированных процессов *PWTBL*

Процесс	Ресурс
1	1
2	2
3	4

Равенство $J = K$ означает, что существует замкнутая цепь взаимоисключающих и ожидающих процессов, то есть выполняются все четыре условия существования тупика.

Модель Холта для описанного примера приведена на рисунке 5.7. На рисунке пронумерованы дуги запросов, которые процессы последовательно генерировали в соответствии с примером. Из рисунка сразу видно, что в результате такой последовательности запросов образовалась замкнутая цепочка: (5, 1, 6, 2, 7, 4), что и говорит о существовании тупика.

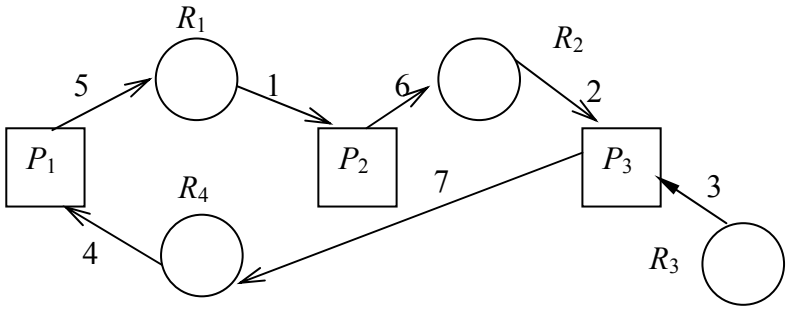


Рисунок 5.7 - Граф распределения ресурсов

Вопросы и задания для самоконтроля

1 Назовите элементы вычислительной схемы асинхронного процесса?

2 Что определяют граф потока данных и граф управления вычислительной схемы?

3 Что называется последовательностью исполнения вычислительной схемы?

4 Какая вычислительная схема называется детерминированной? Приведите пример.

5 Составьте вычислительную схему, содержащую условие соперничества операторов?

6 При каком условии процесс P_i системы будет заблокированным в состоянии S_l; находиться в тупике в состоянии S_l?

7 Какое состояние системы называется безопасным, тупиковым?

8 Для системы, представленной на рисунке 5.8, выполнить задания:

а) составить формальное описание системы;

б) привести примеры допустимых последовательностей изменений состояний;

в) определить тупиковые, опасные и безопасные состояния.

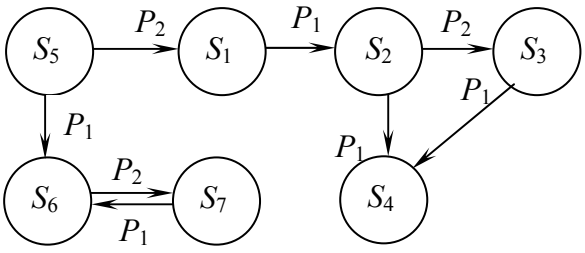


Рисунок 5.8 – Система <σ, π> на семь состояний

9 Построить пространство состояний системы, состоящей из двух процессов, использующих три ресурса. Состояния процессов заданы таблицей 5.4. Определить опасные и тупиковые состояния системы.

Таблица 5.4 – Состояния процессов системы

	Процесс P_1			Процесс P_2		
	R_1	R_2	R_3	R_1	R_2	R_3
Захвачен в состоянии	3	5	4	3	2	4
Освобожден в состоянии	5	6	5	4	5	7

10 Опишите модель Холта (повторно используемых ресурсов).

11 В чем сущность алгоритма обнаружения тупика посредством редукции графа повторно используемых ресурсов?

12 Для модели Холта, показанной на рисунке 5.9, назначьте количество единиц каждого ресурса так, чтобы ни один процесс не был в тупике; только один процесс был в тупике; более одного процесса были в тупике.

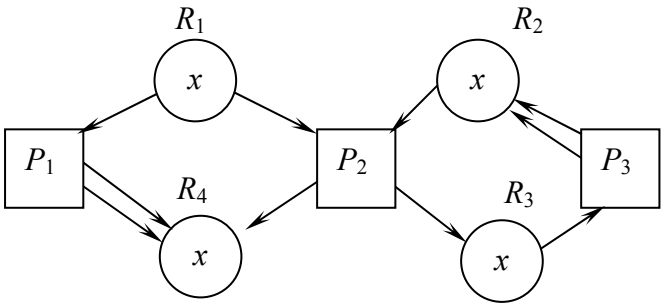


Рисунок 5.9 – Модель Холта процесса

13 Сформулируйте необходимое, достаточное, необходимо и достаточное условия существования тупика в модели Холта.

14 Изложите алгоритм обнаружения тупика по замкнутой цепочке запросов.

Тесты проверки усвоения материала

1 Граф потока данных и граф управления задают:

а) модель Холта;

б) сеть Петри;
в) вычислительную схему;
г) модель пространства состояний системы;
д) модель Маллера.

2 В графе потока данных вычислительной схемы регистры данных изображаются:
а) кружками;
б) прямоугольниками;
в) стрелками;
г) линиями;
д) маркерами.

3 В графе потока данных вычислительной схемы кружки представляют:
а) операторы;
б) регистры данных;
в) управляющие счетчики;
г) условия;
д) переходы.

4 В вычислительной схеме, представленной на рисунке 5.1, определены операторы:
а) 2 и 5;
б) 3 и 6;
в) 1 и 2;
г) 1 и 4;
д) 4 и 5.

5 В вычислительной схеме, представленной на рисунке 5.1, четвертый регистр определяет входные данные для операторов:
а) 3 и 4;
б) 4;
в) 3;
г) 2 и 5;
д) 3 и 6.

6 Допустимая последовательность исполнения вычислительной схемы, представленной на рисунке 5.1:
а) S_4, S_1, S_5, S_2, S_6, S_1;
б) S_4, S_5, S_6, S_1, S_2, S_3;
в) S_1, S_4, S_2, S_5, S_3, S_6;
г) S_6, S_5, S_4, S_3, S_2, S_1;
д) S_6, S_3, S_5, S_2, S_4, S_1.

7 В вычислительной схеме, представленной на рисунке 5.1, третий оператор соперничает за регистр с операторами:

а) 2, 5 и 6;

б) 2 и 6;

в) 5 и 6;

г) 2;

д) 2 и 5.

8 Для модели пространства состояний системы, представленной на рисунке 5.8, справедливо:

а) $S_1 \xrightarrow{*} S_6$;

б) $S_1 \xrightarrow{*} S_4$;

в) $S_2 \xrightarrow{*} S_5$;

г) $S_3 \xrightarrow{*} S_6$;

д) $S_4 \xrightarrow{*} S_6$.

9 Для модели пространства состояний системы, представленной на рисунке 5.8, справедливо:

а) $P_2(S_1) = \{S_2\}$;

б) $P_1(S_6) = \{S_7\}$;

в) $P_1(S_7) = \{S_6\}$;

г) $P_2(S_3) = \{S_2\}$;

д) $P_2(S_5) = \{S_6\}$.

10 В модели пространства состояний системы, представленной на рисунке 5.8, безопасными являются состояния:

а) 6 и 7;

б) 1 и 5;

в) 5;

г) 3 и 4;

д) 7.

11 В модели пространства состояний системы, представленной на рисунке 5.8, состояния 4 и 3 являются:

а) тупиковыми;

б) опасными;

в) безопасными;

г) фиксированными;

д) устойчивыми.

12 Состояние S_i, у которого для всех S_k, таких что $S_i \xrightarrow{*} S_k$, S_k не является тупиковым состоянием, называется:

а) тупиковым;

б) опасным;

в) безопасным;
г) фиксированным;
д) устойчивым.

13 В модели Холта прямоугольниками изображаются:
а) процессы;
б) ресурсы;
в) счетчики;
г) регистры данных;
д) операторы.

14 В модели Холта дуга, ведущая от процесса к ресурсу указывает на то, что процесс:
а) захватывает ресурс;
б) владеет ресурсом;
в) запрашивает ресурс;
г) освобождает ресурс;
д) потребляет ресурс.

15 В модели Холта дуга, ведущая от ресурса к процессу указывает на то, что ресурс:
а) захватывается процессом;
б) выделен процессу;
в) запрашивается процессом;
г) освобождается процессом;
д) потребляется процессом.

16 В модели Холта, представленной на рисунке 5.10, в текущий момент может редуцировать процесс:
а) 1;
б) 2;
в) 1 и 3;
г) 2 и 3;
д) 3.

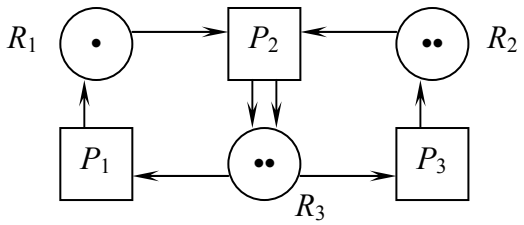


Рисунок 5.10 – Текущее состояние системы

17 В модели Холта, представленной на рисунке 5.10, текущее состояние является тупиком для процессов:
а) 1;
б) 1 и 2;

в) 2;
г) 1 и 3;
д) 2 и 3.

18 Цикл в графе повторно используемых ресурсов системы является:
а) необходимым условием тупика;
б) достаточным условием тупика;
в) необходимым и достаточным условием тупика;
г) необходимым условием отсутствия тупика;
д) достаточным условием отсутствия тупика.

19 Если состояние вычислительной системы фиксированное, то наличие узла в графе повторно используемых ресурсов является:
а) необходимым условием тупика;
б) достаточным условием тупика;
в) необходимым и достаточным условием тупика;
г) необходимым условием отсутствия тупика;
д) достаточным условием отсутствия тупика.

20 Необходимым и достаточным условием тупика является:
а) наличие узла в модели Холта единичной емкости;
б) наличие цикла в модели Холта;
в) наличие цикла в модели Холта единичной емкости;
г) наличие узла в фиксированном состоянии модели Холта;
д) наличие цикла в фиксированном состоянии модели Холта.

6 Сети Петри

6.1 Описание модели

Среди многих существующих методов описания и анализа параллельных систем значительное место занимают сетевые модели, восходящие к сетям специального вида, предложенным в 1962 году Карлом Петри для моделирования асинхронных информационных потоков в системах преобразования данных /14, 15/.

В сети Петри связь между событиями выражается с помощью некоторых условий. Условие может соответствовать таким ситуациям, как наличие данного для операции в программе, состояние некоторого регистра в устройстве ЭВМ и т.п. В сетях Петри условия и события представляются абстрактными объектам. Согласно Петри, событие может наступить, если выполнены все условия, от которых зависит его наступление, а если событие наступило, то изменяется значение некоторого числа условий. Связь между событиями и условиями изображается в виде двудольного ориентированного мультиграфа с двумя типами вершин: кружочками и планками. Кружочки (позиции) соответствуют

условиям, а планки (переходы) – событиям. Дуги могут соединять только кружочки с планками и планки с кружочками. Дуга, ведущая от кружочка к планке, задает входное условие данного события, а дуга от планки к кружочку – выходное условие события. Если условие выполняется, то внутри соответствующего кружочка помещается точка (фишка, маркер). В этом случае говорят, что условие маркировано (размечено). Если условие выполняется с некоторой емкостью n, то внутри соответствующего кружочка помещается n маркеров или число n.

Пример 6.1. Граф сети Петри N_1 приведен на рисунке 6.1. В данной сети пять переходов t_1, t_2, ..., t_5 и пять позиций p_1, p_2, ..., p_5. К примеру, переход t_2 имеет два входных условия p_1 и p_4 с весом дуг, равным 2. Выходным условием перехода t_2 является позиция p_3 с весом дуги – 1. Условие p_4 выполнено с емкостью 2, а условие p_2 не размечено.

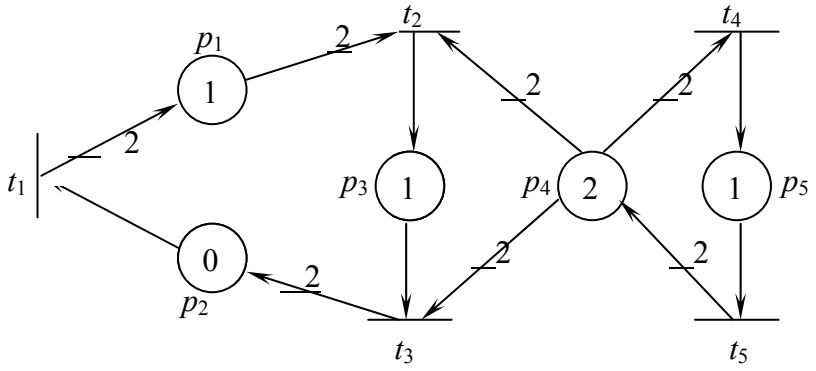


Рисунок 6.1 – Граф сети N_1

6.2 Формальное определение сети Петри

Определение 6.1. Сетью Петри N называется пятерка вида:

$$N = (P, T, F, H, M_0),$$

где $P = \{p_1, p_2, ..., p_m\}$ - конечное непустое множество позиций, $m > 0$;

$T = \{t_1, t_2, ..., t_n\}$ - конечное непустое множество переходов, $n > 0$;

$P \cap T = \varnothing$;

$F: P \times T \to N \cup \{0\}$ - входная функция;

$H: T \times P \to N \cup \{0\}$ - выходная функция;

$M_0: P \to N \cup \{0\}$ - начальная разметка.

Определение 6.2. Разметкой M сети N называется функция M, задающая отображение множества всех позиций P сети во множество натуральных чисел N (включая ноль), т.е. $M: P \to N \cup \{0\}$.

На графе сети Петри кружок, соответствующий позиции p_i, содержит $M(p_i)$ фишек; если $F(p_j, t_i) > 0$, то на графе от позиции p_j к переходу t_i ведет дуга весом $F(p_j, t_i)$; если $H(t_i, p_j) > 0$, то от перехода t_i к позиции p_j ведет дуга весом $H(t_i, p_j)$, где $1 \le i \le n$, $1 \le j \le m$.

Пример 6.2. В сети Петри N_1 (рисунок 6.1) $M(p_4)$ =2, $M(p_2)$ = 0, $F(p_1, t_2)$ = 2, $H(t_2, p_3)$ = 1, $F(p_3, t_2)$ = 0, $H(t_2, p_1)$ = 0 и т.д.

6.3 Правила функционирования сетей Петри

Сеть Петри функционирует, переходя от одной разметки к другой посредством запусков разрешенных переходов.

Определение 6.3. Переход $t \in T$ сети N разрешен при разметке M, если выполняется условие:

$$M(p) \geq F(p, t), \forall\ p \in P. \tag{6.1}$$

Т.е. переход может запускаться в том случае, если каждая из его входных позиций содержит число фишек, не меньшее, чем число дуг, ведущих из этой позиции в переход (или равное весу входной дуги).

Определение 6.4. Множество переходов $T' \subseteq T$ сети N совместно разрешено при разметке M, если выполняется условие:

$$(M(p) \geq F(p, t_i)) \,\&\, (M(p) - \sum_{t_i \in T'} (F(p, t_i) - H(t_i, p))) \geq 0, \forall p \in P, \forall t_i \in T'. \tag{6.2}$$

Определение 6.5. Запуском (срабатыванием) перехода $t \in T$ называется смена разметки M, при которой он разрешен, разметкой M' по правилу:

$$M'(p) = M(p) - F(p, t) + H(t, p), \forall\ p \in P. \tag{6.3}$$

Т.е. при срабатывании разрешённого перехода t из всех его входных позиций p_j ($F(p_j, t) > 0$) удаляется число фишек, равное весу дуги, соединяющей p_j и t, а во все его выходные позиции p_k ($H(t, p_k) > 0$) добавляется число фишек, равное весу дуги, соединяющей t с p_k.

В этом случае говорят, что разметка M' непосредственно следует за разметкой M по переходу t, обозначается: $M \xrightarrow{t} M'$.

Определение 6.6. Запуском (срабатыванием) множества переходов $T' \subseteq T$ называется смена разметки M, при которой T' совместно разрешены, разметкой M' по правилу:

$$M'(p) = M(p) - \sum_{t_i \in T'} F(p, t_i) + \sum_{t_i \in T'} H(t_i, p), \forall\ p \in P. \tag{6.4}$$

В этом случае говорят, что разметка M' непосредственно следует за разметкой M по множеству переходов T', обозначается: $M \xrightarrow{T'} M'$.

Запуски переходов могут осуществляться до тех пор, пока существует хотя бы один разрешенный переход.

Определение 6.7. Разметка сети, при которой не может сработать ни один переход, называется тупиковой разметкой сети.

Пример 6.3. В сети Петри N_1 (рисунок 6.1) не может сработать переход t_1, т.к. его входное условие p_2 не размечено ($M(p_2)=0$), и переход t_2, т.к. в его входной позиции p_1 находится 1 фишка, что меньше веса дуги, соединяющей p_1 с переходом. Остальные переходы могут сработать. Так при запуске перехода t_3 новая разметка сети будет иметь вид: $M'(p_1) = 1$, $M'(p_2) = 2$, $M'(p_3) = 0$, $M'(p_4) = 0$, $M'(p_5) = 1$ или вектор $M'(1, 2, 0, 0, 1)$.

6.4 Дерево разметок сети

Определение 6.8. Разметка M' достижима из разметки M, если существует последовательность разметок $M_1, M_2, \ldots, M'$ и слово $\tau = t_1 t_2 \ldots t_k$ в алфавите T такие, что $M \xrightarrow{t_1} M_1 \xrightarrow{t_2} M_2 \ldots \xrightarrow{t_k} M'$. Обозначается: $M \overset{\tau}{\Rightarrow} M'$.

Слово τ называется последовательностью срабатываний переходов, ведущей от M к M'.

Введем обозначения:

- $R(N, M) = \{M' \mid M \Rightarrow M'\}$ - множество всех достижимых в сети N разметок от разметки M;

- $R(N) = \{M' \mid M_0 \Rightarrow M'\}$ - множество всех достижимых в сети N разметок от начальной разметки M_0, т.е. множество всех достижимых разметок сети.

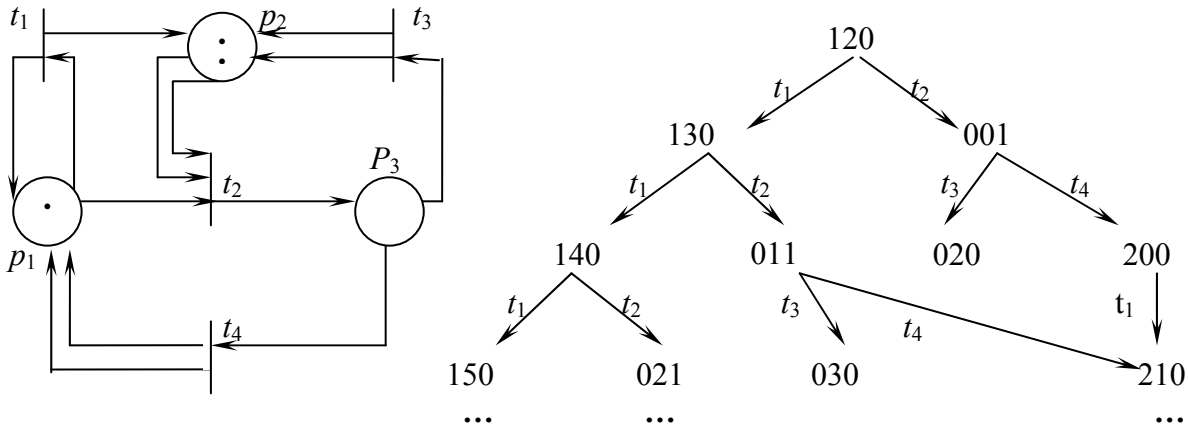
Изменение разметок сети N , происходящее в ходе срабатывания переходов, можно представить в виде дерева разметок – ориентированного графа, множество вершин которого образовано множеством достижимых в сети разметок $R(N)$. Корнем дерева является начальная разметка сети M_0, а от вершины M к вершине M' ведет дуга, помеченная символом перехода t, тогда и только тогда, когда $M \xrightarrow{t} M'$.

Т.к. множество $R(N)$ в общем случае бесконечно, то дерево разметок сети Петри может быть бесконечным.

Пример 6.4. Пусть сеть N_2 задана графом на рисунке 6.2 а. Рисунок 6.2 б содержит фрагмент дерева разметок сети N_2. На дереве разметки (020) и (030) являются тупиковыми. Троеточие указывает на незаконченность данной ветви дерева.

6.5 Свойства сетей Петри

Определение 6.9. Позиция $p \in P$ сети Петри N является k-ограниченной ($k \geq 1$), если для любой достижимой в сети разметки $M \in R(N)$ выполняется условие $M(p) \leq k$. Позиция называется ограниченной, если она является k-ограниченной для некоторого целого значения k. Сеть Петри ограниченна, если все ее позиции ограниченны.



а) Граф сети б) Фрагмент дерева разметок сети

Рисунок 6.2 – Сеть Петри N_2

Определение 6.10. Позиция $p \in P$ сети Петри N называется безопасной, если она является 1-ограниченной. Сеть Петри безопасна, если безопасны все ее позиции.

Определение 6.11. Сеть Петри N является сохраняющей, если сумма фишек во всех ее позициях остается постоянной при работе сети, т.е.:

$$\forall M_1, M_2 \in R(N): \sum_{p \in P} M_1(p) = \sum_{p \in P} M_2(p).$$

Определение 6.12. Для сети Петри N выделяют следующие уровни активности переходов.

Уровень 0. Переход t обладает активностью уровня 0 и называется мёртвым, если он не может сработать ни при одной достижимой в сети разметке $M \in R(N)$.

Уровень 1. Переход t обладает активностью уровня 1 и называется потенциально мёртвым при разметке $M \in R(N)$, если при любой разметке $M' \in R(N, M)$ переход t не может сработать. Разметка M при этом называется t-тупиковой. Разметка M, являющаяся t-тупиковой для всех переходов сети называется тупиковой разметкой сети.

Уровень 2. Переход t обладает активностью уровня 2 и называется потенциально живым при разметке M, если существует разметка $M' \in R(N, M)$ такая, что t разрешён в M'.

Уровень 3. Переход t обладает активностью уровня 3 и называется живым в сети, если для всякой разметки $M \in R(N)$ переход t является потенциально живым при разметке M.

Сеть Петри называется живой, если все её переходы являются живыми.

Определение 6.13. Переход t называется устойчивым в сети N, если t может сработать и никакой другой переход не может, сработав, лишить его этой возможности, т.е.

$$\forall t' \in T \setminus \{t\}, \forall M \in R(N), \forall p \in P: (M(p) \geq F(p,t) \wedge M(p) \geq F(p,t')) \Rightarrow$$
$$\Rightarrow (M(p) \geq F(p,t) + F(p,t')).$$

Задача достижимости. Для данной сети Петри N и разметки M выяснить, принадлежит ли разметка M множеству $R(N)$.

Задача покрываемости. Для данной сети Петри N и разметки M определить, существует ли достижимая разметка $M' \in R(N)$ такая, что $M' \geq M$.

(Отношение $M' \geq M$ истинно, если каждый элемент маркировки M' не меньше соответствующего элемента маркировки M).

Определение 6.14. Сети Петри N_1 и N_2 эквивалентны, если справедливо равенство $R(N_1) = R(N_2)$.

6.6 Анализ сетей Петри

Проблема ограниченности и безопасности сети

Некоторая позиция p сети будет неограниченной если выполнится условие:

$$(M_0 \overset{\tau_1}{\Rightarrow} M_1) \wedge (M_1 \overset{\tau_2}{\Rightarrow} M_2) \wedge (M_1 \leq M_2) \wedge (M_1(p) < M_2(p)). \qquad (6.5)$$

Пример 6.5. В сети Петри N_3 (рисунок 6.3) неограниченной является позиция p_2, т.к. $(M_0=(2000) \overset{t_1}{\rightarrow} M_1=(2100)) \wedge (M_0 \leq M_1) \wedge (M_0(p_2)=0 < M_1(p_2)=1)$.

В рассмотренном примере позиция p_4 также неограниченна, хотя и условие (6.5) не выполняется. Но реализация последовательности срабатываний вида $t_1^n t_2 t_3^n$ ведет к накоплению фишек в p_4. Отличие неограниченности p_4 от неограниченности p_2 состоит в том, что в p_2 может быть бесконечное число фишек, а в p_4 – сколь угодно большое, но обязательно конечное число фишек, т.к. t_3 может сработать конечное число раз, не большее, чем число срабатываний t_1. Неограниченность p_4 вторична по отношению к неограниченности p_2.

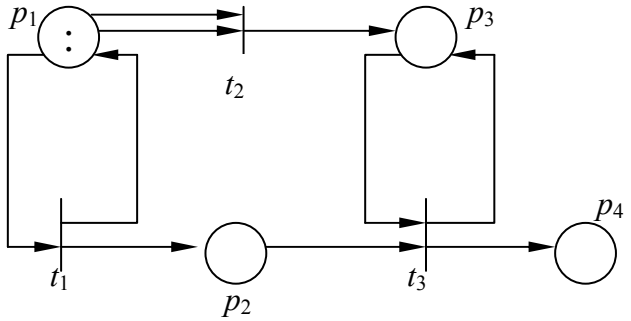


Рисунок 6.3 – Граф сети Петри N_3

Исследование проблемы ограниченности сети сводится к построению покрывающего дерева сети Петри. Для любой сети такой граф всегда конечен и может быть построен с помощью следующего алгоритма.

Алгоритм построения покрывающего дерева сети

1 Первоначально предполагаем, что дерево содержит одну вершину - корень M_0 и не имеет дуг.

2 Пусть M – вершина дерева, которая еще не объявлена листом, но в дереве нет исходящих из нее дуг (граничная вершина). Возможны четыре варианта:

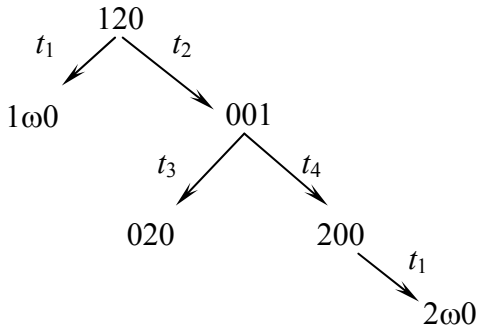
а) если ни один из переходов сети не может сработать при разметке M, то данная вершина дерева называется терминальной и объявляется листом;

б) если на пути от корня дерева к разметке M существует разметка $M' = M$, тогда данная вершина называется дублирующей и объявляется листом;

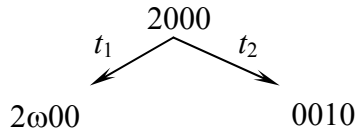
в) если на пути от корня дерева к разметке M существует разметка $M' \leq M$, тогда для каждой позиции p, для которой $M'(p) < M(p)$, значение соответствующей компоненты M заменяется на ω и вершина M объявляется ω-листом (символ ω используется для обозначения бесконечного множества значений маркировки позиции);

г) если не выполнено ни одно из условий а)-в), то M – внутренняя вершина дерева; значит, на дерево добавляется новая вершина M' в результате срабатывания разрешенного перехода t и дуга, ведущая из M в M', помеченная символом t. Вершина M' становится граничной.

Пример 6.6. На рисунке 6.4 а показано покрывающее дерево сети Петри N_2 (рисунок 6.2), на рисунке 6.4 б – покрывающее дерево сети N_3 (рисунок 6.3).



а) покрывающее дерево сети N_2



а) покрывающее дерево сети N_3

Рисунок 6.4 – Примеры покрывающего дерева сети Петри

Важнейшим свойством алгоритма построения покрывающего дерева сети является то, что он за конечное число шагов заканчивает работу. Доказательство основано на трёх леммах.

Лемма 6.1. В любом бесконечном направленном дереве, в котором каждая вершина имеет только конечное число непосредственно последующих вершин, существует бесконечный путь, исходящий из корня.

Доказательство. Пусть x_0 - корневая вершина. Поскольку имеется только конечное число непосредственно следующих за x_0 вершин и общее число вершин в дереве бесконечно, то по крайней мере, одна из непосредственно следующих за x_0 вершин должна быть корнем бесконечного поддерева. Выберем вершину x_1 непосредственно следующую за x_0 и являющуюся корнем бесконеч-

ного поддерева. Теперь одна из непосредственно следующих за ней вершин также является корнем бесконечного поддерева, выберем в качестве такой вершины - x_2. Если продолжать этот процесс бесконечно, то получим бесконечный путь в дереве – $x_0, x_1, x_2, \ldots, x_n, \ldots$. Что и требовалось доказать.

Лемма 6.2. Всякая бесконечная последовательность неотрицательных целых чисел содержит бесконечную неубывающую подпоследовательность.

Доказательство. Возможны два случая.

1 Какой-либо элемент последовательности встречается бесконечно часто, например, x_0. Тогда бесконечная подпоследовательность $x_0, x_0, \ldots, x_0, \ldots$ является бесконечной неубывающей подпоследовательностью.

2 Никакой элемент не встречается бесконечно часто, тогда каждый элемент встречается только конечное число раз. Пусть x_0 - произвольный элемент последовательности. Существует самое большее x_0 целых неотрицательных и меньших x_0 (0, 1, ..., x_0 -1), причем каждый из них присутствует в последовательности только конечное число раз. Следовательно, продвигаясь достаточно долго по последовательности, мы должны встретить элемент x_1, такой, что $x_1 \geq x_0$. Аналогично должен существовать в последовательности x_2, такой, что $x_2 \geq x_1$, и т. д. Это определяет бесконечную неубывающую последовательность $x_0, x_1, x_2, \ldots, x_n, \ldots$

Таким образом, в обоих случаях бесконечная неубывающая подпоследовательность существует. Что и требовалось доказать.

Лемма 6.3. Всякая бесконечная последовательность n-мерных векторов над расширенным символом ω множеством неотрицательных целых чисел содержит бесконечную неубывающую подпоследовательность.

Доказательство. Доказываем индукцией по n, где n - размерность векторного пространства.

1 Базовый случай (n=1). Если в последовательности имеется бесконечное число векторов вида (ω), то они образуют бесконечную неубывающую последовательность (так как справедливо $\omega \leq \omega$). В противном случае бесконечная последовательность, образованная удалением конечного числа экземпляров $<\omega>$, имеет по лемме 6.2 бесконечную неубывающую подпоследовательность.

2 Индуктивное предположение: допустим, что лемма верна для n и докажем её справедливость для n+1. Рассмотрим первую координату. Если существует бесконечно много векторов, имеющих в качестве первой координаты ω, тогда выберем эту бесконечную подпоследовательность, которая не убывает (постоянна) по первой координате. Если только конечное число векторов имеют ω в качестве первой координаты, то рассмотрим бесконечную последовательность целых, являющихся значениями первых координат. По лемме 6.2 эта последовательность имеет бесконечную неубывающую подпоследовательность. Она определяет бесконечную последовательность векторов, которые не убывают по своей первой координате.

В любом случае мы имеем последовательность векторов, неубывающих по первой координате. Применим индуктивное предположение к последовательности n-мерных векторов, которая получается в результате отбрасывания

первой компоненты n+1-мерных векторов. Полученная таким образом бесконечная подпоследовательность является неубывающей по каждой координате.

Следовательно, доказана истинность леммы для любого целого n.

Теорема 6.1. Покрывающее дерево сети Петри конечно.

Доказательство. Докажем методом от противного. Допустим, что дерево достижимости бесконечно. Тогда по лемме 6.1 (и так как число вершин, следующих за каждой вершиной в дереве, ограничено числом переходов n) в нём имеется бесконечный путь x_0, x_1, x_2, …, исходящий из корня x_0. Тогда $M(x_0)$, $M(x_1)$, $M(x_2)$,… - бесконечная последовательность n-мерных векторов над $N \cup \{\omega\}$, а по лемме 4.3 она имеет бесконечную неубывающую подпоследовательность $M(x_{k0}) \le M(x_{k1}) \le M(x_{k2}) \le \ldots$ Но по правилам построения покрывающего дерева $M(x_i) \ne M(x_j)$ для $i \ne j$, поскольку тогда одна из вершин была бы дублирующей и не имела следующих за собой вершин. Следовательно, это бесконечная строго возрастающая последовательность $M(x_{k0}) < M(x_{k1}) < M(x_{k2}) \le \ldots$ Так как $M(x_i) < M(x_j)$, то нам следовало бы заменить по крайней мере одну компоненту $M(x_j)$, не являющуюся ω, на ω. Таким образом, $M(x_{k1})$ имеет по крайней мере одну компоненту, являющуюся ω, $M(x_{k2})$ имеет по крайней мере две ω-компоненты, а $M(x_{kn})$ имеет по крайней мере n ω-компонент. Поскольку маркировки n-мерные, то $M(x_{kn})$ имеет во всех компонентах ω. Но тогда $M(x_{kn+1})$ не может быть больше $M(x_{kn})$. Пришли к противоречию, что доказывает теорему.

Теорема 6.2. Проблема ограниченности сети Петри разрешима.

Алгоритм распознавания ограниченности сети состоит в построении покрывающего дерева и последовательном просмотре конечного числа вершин-разметок. Если будет найдена хотя бы одна разметка с символом ω, то сеть неограниченна, в противном случае – ограниченна.

Покрывающее дерево позволяет найти только неограниченность первого рода. Чтобы найти «вторичную» неограниченность надо построить полное покрывающее дерево сети по следующему алгоритму.

Алгоритм построения полного покрывающего дерева сети

1 Строится покрывающее дерево сети N. Если это дерево не имеет ω-листов, то построение полного покрывающего дерева закончено, и оно совпадает с покрывающим деревом.

2 Если покрывающее дерево содержит ω-лист M, то строится новое покрывающее дерево для сети N', полученной из N заменой начальной разметки M_0 на разметку M. При этом правила срабатывания переходов и изменения разметки сети обобщаются на случай ω-векторов с учетом арифметических свойств расширенного множества натуральных чисел $N \cup \{\omega\}$ ($\omega \pm n = \omega$; $n < \omega$; $\omega \le \omega$, $\forall n \in N$).

3 Построенное новое дерево присоединяется к основному дереву совмещением корня M в новом дереве с листом M в основном. Процесс построения продолжается до тех пор, пока на дереве не исчезнет последний ω-лист.

Теорема 6.3. Проблема ограниченности некоторой позиции p в произвольной сети Петри разрешима.

Из построения полного покрывающего дерева для сети N легко установить, что это дерево конечно. Произвольная позиция p будет неограниченна, если и только если среди вершин полного покрывающего дерева существует вершина M, у которой в позиции p стоит символ ω.

Пример 6.7. Полное покрывающее дерево сети Петри N_3 (рисунок 6.3) показано на рисунке 6.5. В ходе построения покрывающего дерева (пример 6.6.) обнаружена неограниченность первого рода в позиции p_2, а при построении полного покрывающего дерева выявлена неограниченность второго рода в позиции p_4.

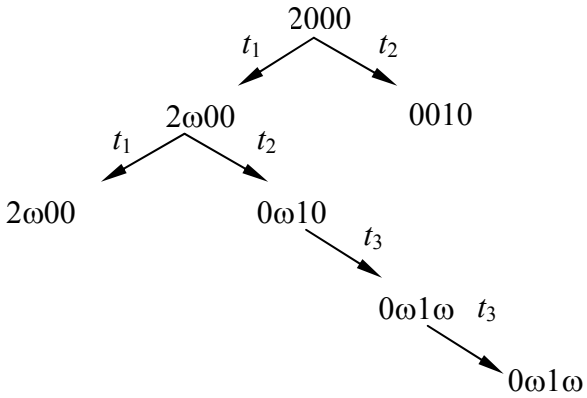


Рисунок 6.5 – Полное покрывающее дерево сети Петри N_3

Теорема 6.4. Существует алгоритм, с помощью которого можно установить для произвольной разметки M и для произвольного перехода t в сети Петри, является ли разметка M t-тупиковой.

В сети N необходимо заменить начальную разметку M_0 на разметку M, после чего построить полное покрывающее дерево для полученной сети. Разметка M будет t-тупиковой тогда и только тогда, когда построенное полное покрывающее дерево не содержит дуги, помеченной символом t.

Теорема 6.5. Проблема безопасности сети Петри разрешима.

Сеть с n позициями безопасна тогда и только тогда, когда все вершины ее покрывающего дерева представляют собой векторы, состоящие только из 0 и 1.

Теорема 6.6. Проблема потенциальной живости переходов сети при разметке M разрешима.

Переход t потенциальной живой при разметке M тогда и только тогда, когда он метит некоторую дугу в полном покрывающем дереве сети хотя бы один раз.

Теорема 6.7. Существует алгоритм, с помощью которого можно узнать, получит ли данная позиция в сети хотя бы одну фишку.

Если хотя бы в одной вершине покрывающего дерева позиция, соответствующая месту p, содержит число $n>0$, то место p может получить фишку в процессе функционирования сети.

Теорема 6.8. Существует алгоритм, с помощью которого можно узнать, может ли данный переход сработать сколь угодно большое число раз.

Достаточно присоединить к данному переходу новое «висячее» выходное место p и затем выяснить, является ли оно неограниченным.

Теорема 6.9. Проблема сохранения сети Петри разрешима.

Так как полное покрывающее дерево конечно, то для каждой вершины можно вычислить сумму значений ее позиций. Если эта сумма одинакова для каждой достижимой маркировки, то сеть Петри является сохраняющей. Если маркировка некоторой позиции совпадает с ω, то эта позиция должна был исключена из рассмотрения.

Теорема 6.10. Проблема покрываемости сети Петри разрешима.

Задача решается путём простого перебора вершин дерева достижимости. При этом для вершины M ищется вершина M' такая, что $M' \geq M$. Если такой вершины не существует, то разметка M не является покрываемой. Если она найдена, то M' определяет покрывающую маркировку для M. Если компонента маркировки M', соответствующая некоторой позиции p совпадает с ω, то конкретное её значение может быть вычислено. В этом случае на пути от начальной маркировки к покрывающей маркировке имеется повторяющаяся последовательность переходов, запуск которой увеличивает значение маркировки в позиции p. Число данных повторений должно быть таким, чтобы значение маркировки в позиции p превзошло или сравнялось с $M(p)$.

Теорема 6.11. Проблема достижимости некоторой разметки сводится к проблеме живости сети и наоборот.

Данная проблема пока в общем случае алгоритмически не разрешима.

6.7 Матричный подход к анализу сетей Петри

Введем две матрицы и вектор:

1) $D^I(i, j) = F(p_j, t_i), 1 \leq i \leq n, 1 \leq j \leq m$ - матрица входов;

2) $D^O(i, j) = H(t_i, p_j), 1 \leq i \leq n, 1 \leq j \leq m$ - матрица выходов;

3) $M(j) = M(p_j), 1 \leq j \leq m$ - вектор начальной разметки.

Тогда условие срабатывания переходов примет вид:

переход t_i ($1 \leq i \leq n$) сработает при разметке $M \Leftrightarrow M(j) \geq D^I(i, j), 1 \leq j \leq m.$

Правило срабатывания перехода t_i ($1 \leq i \leq n$), разрешенного при разметке M, перепишем в виде:

$$M'(j) = M(j) - D^I(i, j) + D^O(i, j), 1 \leq j \leq m.$$

Определение 6.15. Вектор $X = (x_1, x_2, \ldots, x_n)$, где n – число переходов в сети Петри, называется вектором запуска, если его i-тая координата равна числу запусков перехода t_i в заданной последовательности запусков $t_{j1}, t_{j2}, \ldots, t_{jk}$.

Теорема 6.12. Если разметка M' достижима из разметки M, то существует целое неотрицательное решение уравнения:

$$M' = M + XD, \tag{6.6}$$

где $D = D^O - D^I$ - составная матрица изменений;

X – вектор, относительно которого решается уравнение.

Пример 6.8. Граф сети Петри задан на рисунке 6.6. Найти последовательность срабатываний переходов, ведущую из начальной разметки к разметке $M' = (0, 2, 1, 2)$.

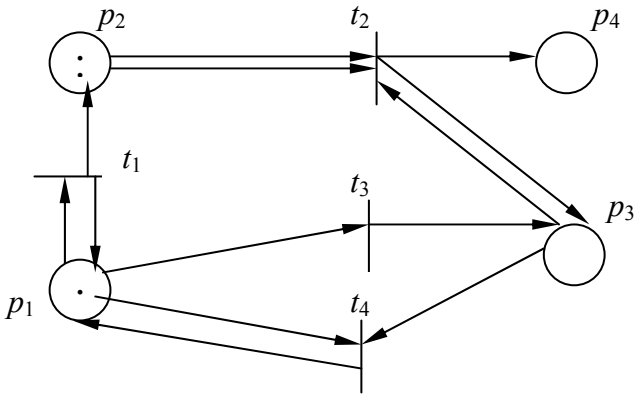


Рисунок 6.6 – Граф сети Петри N_4

Данная сеть описывается следующими матрицами:

$$D^I = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 2 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 \end{pmatrix}; \; D^O = \begin{pmatrix} 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \end{pmatrix}; \; D = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & -2 & 0 & 1 \\ -1 & 0 & 1 & 0 \\ 0 & 0 & -1 & 0 \end{pmatrix}; \; M_0 = (1, 2, 0, 0).$$

Воспользовавшись уравнением (6.6), запишем:

$$(0, 2, 1, 2) = (1, 2, 0, 0) + (x_1, x_2, x_3, x_4) \cdot \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & -2 & 0 & 1 \\ -1 & 0 & 1 & 0 \\ 0 & 0 & -1 & 0 \end{pmatrix}.$$

Решением матричного уравнения будет вектор $X = (4, 2, 1, 0)$. Проверкой устанавливается, что маркировка M' достижима из M последовательностью запусков переходов вида $(t_1, t_1, t_1, t_1, t_3, t_2, t_2)$.

6.8 Моделирование систем на основе сетей Петри

6.8.1 Моделирование последовательных процессов

Рассмотрим, как сетью Петри может быть представлен отдельный процесс. Отдельный процесс описывается программой на одном из существующих языков программирования.

Пример 6.9. Текст программы на языке программирования *Pascal*, вычисляющей произведение всех чётных чисел из отрезка [1, *Y*] для $Y \in N$:

```
program mult;
var X, Y: integer;
begin
  read(Y);
  X:=1;
  while Y>0 do
    begin
      if  Y mod 2 = 0 then X:=X*Y;
      Y:=Y-1;
    end;
  write(X);
end.
```

Программа представляет два различных аспекта процесса: вычисление и управление. Сети Петри удачно представляют структуру и управление программ. Они предназначены для моделирования упорядочения действий и потока информации, а не для действительного вычисления самих значений.

Стандартный способ представления структуры программы и потока управления в ней – схемы алгоритма, которые, в свою очередь, могут быть представлены сетями Петри. Схема алгоритма программы изображена на рисунке 6.7 а, где использованы следующие обозначения блоков:

a: *read*(*Y*); *X*:=1;
b: *Y*>0;
c: *Y mod* 2 =0;
d: *X*:=*X***Y*;
e: *Y*:=*Y*-1;
f: *write*(*X*).

В сети Петри (рисунок 6.7 б), моделирующей схему алгоритма, узлы схемы представляются переходами сети, а дуги - позициями сети Петри. Фишка в сети представляет счетчик команд схемы алгоритма.

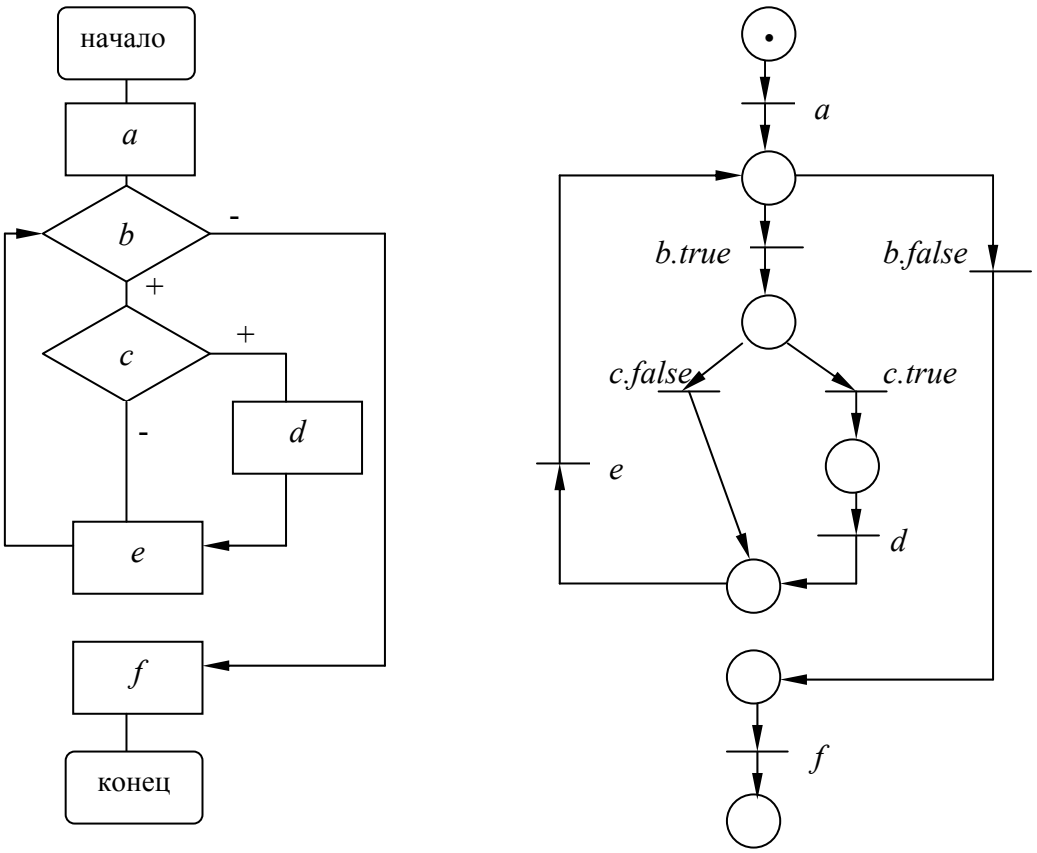


Рисунок 6.7 – Структура программы

6.8.2 Моделирование взаимоисключения параллельных процессов

Сеть Петри на рисунке 6.8 моделирует механизм взаимного исключения для двух процессов P_1 и P_2. Она легко обобщается на произвольное число процессов.

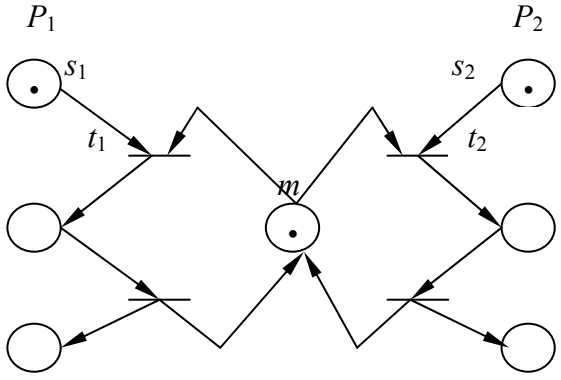


Рисунок 6.8 – Моделирование взаимного исключения процессов P_1 и P_2

Позиция m представляет условие «критическая секция свободна», разрешающее вход в критическую секцию. Попытка процесса $P_1(P_2)$ войти в критическую секцию осуществляется после помещения фишки в его позицию $s_1(s_2)$.

Такая попытка будет успешной, если в позиции m содержится фишка. Если оба процесса пытаются войти в критическую секцию одновременно, то переходы t_1 и t_2 вступят в конфликт, и только один из них сможет запуститься. Запуск t_1 запретит запуск перехода t_2, вынуждая процесс P_2 ждать, пока процесс P_1 выйдет из своей критической секции и возвратит фишку обратно в позицию m.

6.8.3 Моделирование взаимодействия типа «Производитель – Потребитель»

Разделяемым объектом в задаче является буфер, посредством которого реализуется взаимодействие через асинхронную передачу сообщений. Такое взаимодействие может быть промоделировано сетью Петри, изображенной на рисунке 6.9.

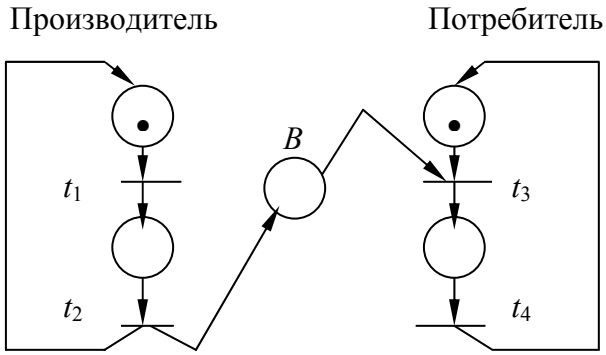


Рисунок 6.9 – Взаимодействие типа «Производитель – Потребитель»

Позиция B сети Петри представляет буфер, каждая фишка соответствует сообщению, которое произведено, но еще не использовано. Переходам сети соответствуют следующие события взаимодействующих процессов:

1) t_1 – процесс «Производитель» генерирует сообщение;

2) t_2 – процесс «Производитель» помещает сообщение в буфер обмена;

3) t_3 – процесс «Потребитель» извлекает сообщение из буфера обмена;

4) t_4 – процесс «Потребитель» использует считанное сообщение.

Вопросы и задания для самоконтроля

1 Какими элементами формально задается сеть Петри?

2 Как строится граф сети Петри?

3 Какой переход сети Петри называется разрешенным?

4 Запишите правило срабатывания перехода сети Петри.

5 Какие условия должны выполняться для совместного срабатывания множества переходов?

6 Запишите правило срабатывания множества совместно возможных переходов сети.

7 Как построить дерево разметок сети Петри?

8 Какая позиция сети называется ограниченной, безопасной?

9 Охарактеризуйте уровни активности переходов сети.

10 Сформулируйте постановку задачи достижимости и покрываемости сети Петри. Каким образом они разрешимы?

11 В чем разница между ограниченностью позиции сети первого и второго рода? Как их обнаружить в сети?

12 Какие типы вершин возможны на покрывающем дереве сети Петри?

13 Постройте дерево разметок, покрывающее и полное покрывающее дерево сети N_5, заданной графом на рисунке 6.10.

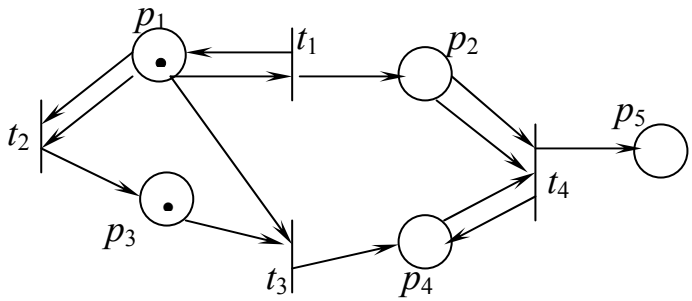


Рисунок 6.10 – Граф сети N_5

14 Исследуйте сеть N_5 на безопасность, ограниченность, сохраняемость. Определите уровни активности ее переходов.

14 Постройте сеть Петри, моделирующую схему алгоритма программы вычисления факториала числа.

15 Приведите пример сети Петри, моделирующей механизм взаимного исключения для трех параллельных процессов.

Тесты проверки усвоения материала

1 Для сети Петри N_6, представленной на рисунке 6.11, справедливо:

а) $F(p_2, t_3)=0$;

б) $M(p_3)=0$;

в) $F(p_3, t_3)=1$;

г) $M(p_1)=2$;

д) $H(t_3, p_3)=2$.

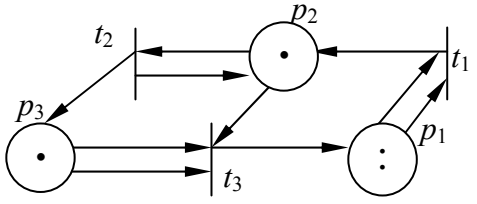


Рисунок 6.11 – Граф сети N_6

2 Переход $t \in T$ сети Петри разрешен при разметке M, если выполняется условие:

а) $M(p) \le F(p, t), \forall\ p \in P$;

б) $M(p) = F(p, t), \forall\ p \in P$;

в) $M(p) \ge F(p, t), \forall\ p \in P$;

г) $M(p) > F(p, t), \forall\ p \in P$;

д) $M(p) < F(p, t), \forall\ p \in P$.

3 В сети Петри N_6, представленной на рисунке 6.11, могут сработать переходы:

а) t_1 и t_3;

б) t_1 и t_2;

в) t_2 и t_3;

г) t_2;

д) t_3.

4 Для сети Петри, представленной на рисунке 6.11, тупиковой будет разметка:

а) 201;

б) 211;

в) 200;

г) 101;

д) 010.

5 При срабатывании перехода $t \in T$ сети Петри текущая разметка M, при которой он разрешен, меняется разметкой M' по правилу:

а) $M'(p) = M(p) - F(p, t) + H(t, p), \forall\ p \in P$;

б) $M'(p) = M(p) + F(p, t) - H(t, p), \forall\ p \in P$;

в) $M'(p) = M(p) - F(p, t) - H(t, p), \forall\ p \in P$;

г) $M'(p) = M(p) + F(p, t) + H(t, p), \forall\ p \in P$;

д) $M'(p) = F(p, t) + H(t, p) - M(p), \forall\ p \in P$.

6 В сети Петри N_6, представленной на рисунке 6.11, при срабатывании перехода t_2 новая разметка будет:

а) 211;

б) 202;

в) 212;

г) 221;

д) 201.

7 Множество переходов $T' \subseteq T$ сети Петри совместно разрешено при разметке M, если выполняется условие:

а) $(M(p) \le F(p, t_i)) \& (M(p) - \sum_{t_i \in T'} (F(p, t_i) - H(t_i, p))) \ge 0, \forall p \in P, \forall t_i \in T'$;

б) $(M(p) \ge F(p, t_i)) \& (M(p) - \sum_{t_i \in T'} (F(p, t_i) + H(t_i, p))) \ge 0, \forall p \in P, \forall t_i \in T'$;

в) $(M(p) \ge F(p, t_i)) \& (M(p) - \sum_{t_i \in T'} (F(p, t_i) - H(t_i, p))) \ge 0, \forall p \in P, \forall t_i \in T'$;

г) $(M(p) \ge F(p, t_i)) \& (M(p) - \sum_{t_i \in T'} (H(t_i, p) - F(p, t_i))) \ge 0, \forall p \in P, \forall t_i \in T'$;

д) $(M(p) > F(p, t_i)) \& (M(p) - \sum_{t_i \in T'} (F(p, t_i) - H(t_i, p))) \ge 0, \forall p \in P, \forall t_i \in T'$.

8 В сети N_6, представленной на рисунке 6.11, переходы t_2 и t_3 будут совместно возможными при разметке:

а) 211;

б) 022;

в) 212;

г) 231;

д) 202.

9 Если на дереве разметок сети Петри каждая вершина состоит только из 0 и 1, то сеть является:

а) безопасной;

б) живой;

в) сохраняющей;

г) покрываемой;

д) устойчивой.

10 Если для сети Петри выполнено условие $\forall M_1, M_2 \in R(N): \sum_{p \in P} M_1(p) = \sum_{p \in P} M_2(p)$, то сеть является:

а) живой;

б) сохраняющей;

в) ограниченной;

г) безопасной;

д) устойчивой.

11 Позиция p сети Петри будет неограниченной, если выполнится условие:

а) $(M_0 \overset{\tau_1}{\Rightarrow} M_1) \wedge (M_1 \overset{\tau_2}{\Rightarrow} M_2) \wedge (M_1 \le M_2) \wedge (M_1(p) = M_2(p))$;

б) $(M_0 \overset{\tau_1}{\Rightarrow} M_1) \wedge (M_1 \overset{\tau_2}{\Rightarrow} M_2) \wedge (M_1 \le M_2) \wedge (M_1(p) > M_2(p))$;

в) $(M_0 \overset{\tau_1}{\Rightarrow} M_1) \wedge (M_0 \overset{\tau_2}{\Rightarrow} M_2) \wedge (M_1 \le M_2) \wedge (M_1(p) < M_2(p))$;

г) $(M_0 \overset{\tau_1}{\Rightarrow} M_1) \wedge (M_0 \overset{\tau_2}{\Rightarrow} M_2) \wedge (M_1 \le M_2) \wedge (M_1(p) > M_2(p))$;

д) $(M_0 \overset{\tau_1}{\Rightarrow} M_1) \wedge (M_1 \overset{\tau_2}{\Rightarrow} M_2) \wedge (M_1 \le M_2) \wedge (M_1(p) < M_2(p))$.

12 Если в процессе построения покрывающего дерева сети Петри получено $M=(1020) \overset{t}{\rightarrow} M'=(1030)$, то разметка M' называется:

а) терминальной;

б) дублирующей;

в) тупиковой;

г) внутренней;

д) ω-листом.

13 Для сети Петри бесконечным может быть:

а) дерево разметок сети;

б) покрывающее дерево сети;

в) граф сети;

г) множество переходов сети;

д) множество позиций сети.

14 Если при любой разметке $M' \in R(N, M)$ переход t не может сработать, то он является:

а) потенциально живым при разметке M;

б) живым в сети N;

в) мертвым в сети N;

г) потенциально мёртвым при разметке M;

д) устойчивым при разметке M.

15 Переход t_3 сети Петри, покрывающее дерево которой представлено на рисунке 6.12, является:

а) живым переходом сети;

б) потенциально мертвым при M_1;

в) потенциально живым при M_1;

г) мертвым переходом сети;

д) потенциально мертвым при M_2.

16 Потенциально живыми при разметке M_1 сети Петри, покрывающее дерево которой представлено на рисунке 6.12, являются переходы:

а) t_1 и t_3;
б) t_1 и t_2;
в) t_3 и t_4;
г) t_1 и t_4;
д) t_2 и t_4.

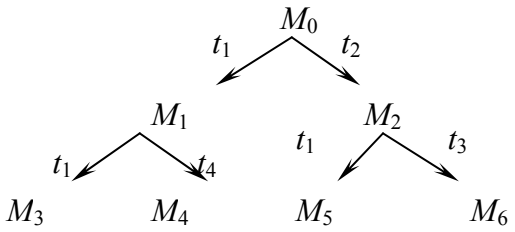


Рисунок 6.12 – Покрывающее дерево сети

17 Для сети Петри N_6, представленной на рисунке 6.11, верным является равенство:

а) $D^I(2, 1)=1$;
б) $D^O(2, 3)=0$;
в) $D^I(3, 3)=1$;
г) $D^O(3, 3)=0$;
д) $M(3)=2$.

18 Для сети Петри N_6, представленной на рисунке 6.11, верным является соотношение:

а) $D^O(3, 3)>M(3)$;
б) $D^O(2, 3)>M(3)$;
в) $D^I(3, 3)=M(3)$;
г) $D^I(2, 1)<M(1)$;
д) $D^I(1, 1)=M(2)$.

19 Сеть Петри игнорирует такой аспект вычислительных процессов, как:

а) структура программы;

б) упорядочение действий;

в) вычисление реальных значений переменных;

г) потоки информации;

д) механизмы взаимоисключения процессов.

20 В сети Петри, моделирующей схему алгоритма, узлы схемы представляются:

а) переходами сети;

б) позициями сети;

в) входными дугами переходов;

г) выходными дугами переходов;

д) фишками.

7 Протоколы и интерфейсы

7.1 Предпосылки теории протоколов

Современное аппаратное и программное обеспечение ЭВМ строится на основе модульного принципа. Организация модулей в систему предполагает взаимодействие между модулями. Фундаментальным понятием, отражающим аспект взаимодействия модулей в системе, является понятие интерфейса.

В широком смысле под интерфейсом (или сопряжением) понимается совокупность правил, обеспечивающих совместное функционирование аппаратных и программных средств. Иногда интерфейсом называют и сами технические средства, созданные в соответствии с этими правилами.

Сопряжение (в отличие от устройств сопряжения) не является частью аппаратных средств, а представляет собой сечение (границу) между модулями системы, на котором определены сигналы и процедуры обмена. Сопряжение точно определяет, какие сигналы или сообщения могут пересекать это сечение и каков их смысл. Оно должно обеспечивать единый метод передачи данных независимо от внутренних особенностей взаимодействующих модулей и относиться исключительно к обмену сигналами между устройствами.

Совокупность правил и средств, составляющих структуру сопряжения - интерфейс - принято делить на три уровня: механический, электрический и логический.

Механический (конструктивный) уровень определяет совокупность требований, которым должны удовлетворять узлы интерфейсной аппаратуры - кабель, линии связи, разъемные соединения, адаптеры и т.п.

Электрический уровень сопряжения оговаривает допуски на параметры входных и выходных электрических сигналов, токовые нагрузки на источник сигнала. Они зависят от принятой элементной базы.

Логический уровень сопряжения обеспечивает координацию протекания информационных потоков между устройствами. Он определяет способ представления сигналов, символов, сообщений, обозначения и классификацию сигналов и сообщений. Логический уровень - наиболее сложная часть правил сопряжения. Логическое согласование объединяемых в систему устройств предусматривает функционирование сопрягаемых модулей по установленным алгоритмам. Основным понятием, отражающим координированное функционирование объектов логического уровня интерфейса, является понятие протокола информационного обмена.

Под протоколом информационного обмена обычно понимают свод правил, которых нужно придерживаться, чтобы обеспечить упорядоченный информационный обмен между двумя или более объектами.

Предметом возникшей теории протоколов (информационного обмена), являются способы задания, анализа и синтеза интерфейсного взаимодействия. При составлении протоколов необходимо удовлетворить несколько требований: компактное представление правил обмена, их однозначное толкование, возможность проверки задания на полноту и непротиворечивость, возможность перехода от протоколов непосредственно к синтезу интерфейсного устройства.

Поставленные задачи обычно решается последовательностью трех крупных этапов:

1) разработка формальной модели, отражающей закон функционирования системы;

2) абстрактный синтез – кодирование состояний и сигналов системы;

3) структурный синтез – построение реализаций в заданном элементном базисе.

Итак, первоначально необходимо выполнить формализацию языка описания. Формализацию будем осуществлять согласно В.И. Варшавскому в терминах асинхронных процессов /9/.

7.2 Композиция асинхронных процессов

Определение 7.1. Репозицией асинхронного процесса $P = <S, F, I, R>$ называется эффективный асинхронный процесс $P' = <S', F', I', R'>$ такой, что $S' \subseteq I \cup R \cup S^{д}$, $I' \subseteq R$, $R' \subseteq I$.

Ситуации S' репозиции могут содержать лишь те ситуации из исходного процесса, которые являются лишь инициаторами или результантами и некоторые дополнительные ситуации из $S^{д}$, отсутствующие в описании исходного АП ($S \cap S^{д} = \varnothing$). Отношение F' задает траектории переходов от элементов из $I' \subseteq R$ к элементам $R' \subseteq I$, возможно через дополнительные ситуации из $S^{д}$.

Определение 7.2. Приведенным асинхронным процессов называется процесс $P^{п} = <S^{п}, F^{п}, I^{п}, R^{п}>$ такой, что $S^{п} = S \cup S^{д}$, $F^{п} = F \cup (F' \setminus (F' \cap (S' \times I)))$, $I^{п} \subseteq I$, $R^{п} = R \cup S^{д}$.

Иными словами, приведенный процесс является объединением АП и его репозиции, в котором из отношения F выброшены пары, задающие переходы к инициаторам процесса (образующие пары $F' \cap (S' \times I)$).

В зависимости от того, какую репозицию имеет процесс P, соответствующий приведенный процесс $P^{п}$ будем называть полностью или частично приведенным. Если для АП репозиции не существует, то он называется неприводимым. Для приведенного процесса со структурированными путем выделения входной и выходной компонент ситуациями понятия результанта и инициатора можно конкретизировать, например, следующим образом. Отнесем ситуацию к результантам, если в ситуациях, принадлежащих всем траекториям, проходящим через данную ситуацию, после нее сохраняется то же значение выходной

компоненты, что и в самой этой ситуации. К инициаторам отнесем те ситуации, которые образованы из результантов такой заменой значений входной компоненты, что полученная ситуация не относится к результантам.

Если некоторый процесс можно рассматривать как модель совместного функционирования нескольких подпроцессов, то структурными единицами (компонентами) ситуаций процесса могут являться ситуации подпроцессов.

Пусть задан неприведенный АП $P = <S, F, I, R>$, ситуации которого структурированы, т. е. множество ситуаций S представимо упорядоченной тройкой $S = (X, Y, Z)$, где X, Y и Z - соответственно множества значений (символов) входной компоненты, выходной компоненты и компоненты, не являющейся ни входной ни выходной.

Образуем p-блочное разбиение π множества S ситуаций процесса P, в ситуациях каждого блока которого входная компонента принимает фиксированное значение $x_j, 1 \le j \le p$. Выберем $r<p$ различных значений входной компоненты (составляющих множество $X^* \subset X$). Ситуации, входящие в те блоки разбиения, которые соответствуют выбранным значениям входной компоненты, составляют подмножество S^*, $S^* \subset S$.

Для каждого инициатора $s_i \in I$ построим множество ситуаций $S(s_i)$, встречающихся на траекториях процесса P, ведущих из указанного инициатора. Образуем множество $S(X^*)$ как объединение тех множеств $S(s_i)$, для которых справедливо $S(s_i) \le S^*$ т. е.

$$S(X^*) = \bigcup_{S(s_i) \subseteq S^*} S(s_i).$$

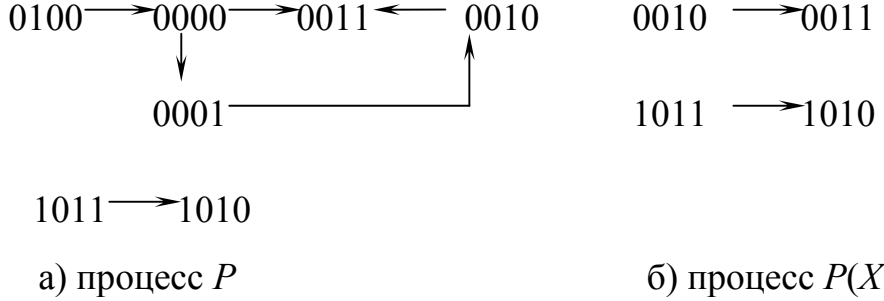
Построим также $F(X^*) = F \cap (S(X^*) \times S(X^*))$, $I(X^*) = I \cap S(X^*)$, $R(X^*) = R \cap S(X^*)$.

Определение 7.3. Назовем процесс $P(X^*) = <S(X^*), F(X^*), I(X^*), R(X^*)>$ редукцией неприведенного процесса $P = <S, F, I, R>$ по выбранному множеству X^* значений входной компоненты.

Аналогично определяется редукция $P(Y^*)$ неприведенного процесса по выбранному множеству Y^* значений выходной компоненты.

Суть операции редукции состоит в сведении данного АП к более простому. Такая операция необходима тогда, когда из полного описания процесса следует выделить некоторую интересующую исследователя часть.

Пример 7.1. Пусть процесс P задан диаграммой (рисунок 7.1 а), в которой звездочки опущены.



а) процесс P б) процесс $P(X^*)$

Рисунок 7.1 – Диаграммы переходов процесса

Здесь S = {0100, 0000, 0010, 0001, 0011, 1011, 1010}, I = {0100, 0010, 1011}, R = {0011, 1010}. Два первых элемента вектора полного состояния выберем в качестве входной компоненты. Тогда редукцией процесса P по множеству X^* = {00, 10} будет процесс $P(X^*)$ с диаграммой полных состоянии на рисунке 7.1 б. Причем S^* = {0000, 0010, 0001, 0011, 1011, 1010}, $S(X^*)$ = {0010, 0011, 1011, 1010}, $I(X^*)$ ={0010, 1011}, $R(X^*)$ = {0011, 1010}.

Рассмотрим теперь репозицию редукции $P(X^*)$ процесса P. Образуем p-блочное разбиение π' ситуаций S' процесса P', каждый блок которого соответствует фиксированному значению x_j ($1 \le j \le p$) входной компоненты. Образуем множество S'^* как объединение тех блоков π', в ситуациях которых $x_j \in X^*$. Далее для каждого инициатора $i'_j \in R(X^*)$ процесса P' и каждого результанта $r'_m \in I(X^*)$ рассмотрим множество ситуаций $S'_k(i'_j, r'_m)$ принадлежащих k-й траектории из i'_j в r'_m.

Построим:

1) множество $S'(X^*) = \bigcup\limits_{S'_k(i', r''_m) \subset S'^*} S_k(i'_j, r'_m)$;

2) отношение $F'(X^*)$, задающее следование ситуаций на этих траекториях, $F'(X^*) = F \cap (S'(X^*) \times S'(X^*))$;

3) множества $I'(X^*) = I' \cap S'(X^*)$, $R'(X^*) = R' \cap S'(X^*)$

Процесс $P'(X^*) = \langle S'(X^*), F'(X^*), I'(X^*), R'(X^*) \rangle$ будет искомой репозицией редукции $P(X^*)$ процесса P.

Аналогично по репозиции P' процесса P строится репозиция редукции $P(Y^*)$. Редукция приведенного процесса $P^{\text{п}}$ строится по редукции неприведенного процесса P и репозиции редукции $P'(X^*)$.

Рассмотрим два АП - процесс $P_1 = \langle S_1, F_1, I_1, R_1 \rangle$ (не обязательно приведенный) и приведенный процесс $P_2^n = \langle S_2, F_2, I_2, R_2 \rangle$, ситуации которых структурированы: в ситуациях P_1 выделена выходная, а в ситуациях P_2 - входная компоненты. Пусть все или некоторые значения выходной компоненты y^1 ситуаций из S_1 соответствуют всем или некоторым значениям входной компоненты x^2 ситуаций из S_2. Обозначим проекции множества пар соответствующих друг другу значений компонент на множества Y_1 и X_2 через Y_1^* и X_2^* соответственно. По множествам Y_1^* и X_2^* построим редукции $P_1(Y_1^*)$ и $P_2^n(X_2^*)$ процессов P_1 и P_2. Пусть $P_2^n(X_2^*)$ - полностью приведенный процесс и $Y_1^{**} = Y_1^*$, $X_2^{**} = X_2^*$.

Построим, если это возможно, процесс $P_3 = \langle S_3, F_3, I_3, R_3 \rangle$, ситуации которого представимы в виде пар $s^3 = (s^1, s^2)$, такой что:

1) $s^1 \in S_1(Y_1^*)$, $s^2 \in S_2(X_2^*)$, т.е. $S_3 \subseteq S_1(Y_1^*) \times S_2(X_2^*)$;

2) выходная компонента y^1 ситуации s^1 равна входной компоненте x^2 ситуации s^2 ($y^1 = x^2$);

3) если в s^3 компонента $s^2 \in I_2(X_2^*)$, то $s^1 \in R_1(Y_1^*)$;

4) если $(s_i^1, s_j^2) F_3 (s_k^1, s_l^2)$, то либо $(s_i^1 F_1 s_k^1) \& (s_j^2 F_2 s_l^2)$, либо $(s_i^1 F_1 s_k^1) \& (s_j^2 = s_l^2)$, либо $(s_i^1 = s_k^1) \& (s_j^2 F_2 s_l^2)$.

Определение 7.4. Назовем последовательной композицией асинхронных процессов P_1 и P_2^n асинхронный процесс P_3, образованный отождествлением

значений входной и выходной компонент ситуаций редуцированных процессов $P_1(Y_1^*)$ и $P_2(X_2^*)$ соответственно при выполнении перечисленных выше ограничений 1) - 4).

Смысл ограничения 1) состоит в том, чтобы при функционировании процесса P_2 в составе процесса P_3 в редукции $P_2^n(X_2^*)$ не встречались ситуации, не принадлежащие множеству $S_2(X_2^*)$. Ограничение 4) говорит о том, что для редукции $P_2^n(X_2^*)$ процесса P_2 в составе P_3 не может возникнуть новых траекторий.

Из определения 7.4 следует, что:

1) поскольку ситуации $s^3 \in S_3$ строятся из пар вида $s^1 = (x^1, y^1, z^1)$, $s^2 = (x^2, y^2, z^2)$, то компоненты $s^3 = (x^3, y^3, z^3)$ определяются как $x^3 = x^1$, $y^3 = y^2$, $z^3 = (z^1, y^1{=}x^2, z^2)$, $S_3 \subseteq S_1(Y_1^*) \times S_2(X_2^*)$, причем проекции S_3 на $S_1(Y_1^*)$ и $S_2(X_2)$ равны соответственно $S_1(Y_1^*)$ и $S_2(X_2)$;

2) $I_3 \subseteq (I_1(Y_1^*) \times R_2(X_2^*)) \cap S_3$, причем проекции I_3 на $I_1(Y_1^*)$ и $R_2(X_2^*)$ равны соответственно $I_1(Y_1^*)$ и $R_2(X_2^*)$;

3) $R_3 \subseteq (R_1(Y_1^*) \times R_2(X_2^*)) \cap S_3$, причем проекции R_3 на $R_1(Y_1^*)$ и $R_2(X_2^*)$ равны соответственно $R_1(Y_1^*)$ и $R_2(X_2^*)$.

Рассмотрим теперь приведенный АП $P^{\text{п}} = \langle S^{\text{п}}, F^{\text{п}}, I^{\text{п}}, R^{\text{п}} \rangle$, соответствующий некоторому процессу P и его репозиции P'. Пусть ситуации процесса $P^{\text{п}}$ структурированы: в них выделены входная x и выходная y компоненты и все или некоторые значения выходной компоненты y ситуаций этого процесса соответствуют всем или некоторым значениям входной компоненты x ситуаций того же процесса. Обозначим проекцию множества эквивалентных пар значений компонент на множества X и Y через X^* и Y^* соответственно. По множествам X^* и Y^* построим редукции $P^{\text{п}}(X^*)$ и $P^{\text{п}}(Y^*)$ процесса P.

Построим автономный процесс $P^3(X^*, Y^*) = \langle S^3, F^3 \rangle$, удовлетворяющий следующим ограничениям:

1) $S^3 \subseteq S^{\text{п}}(X^*) \cap S^{\text{п}}(Y^*)$;

2) выходная компонента y^3 ситуации s^3 эквивалентна входной компоненте x^3 ($y^3{=}x^3$);

3) $F^3 = (F(X^*) \cup F'(X^*)) \cap (F'(Y^*) \cup F(Y))$, где F и F' задают причинно-следственные связи для неприведенного процесса и его репозиции, соответствующих процессу $P^{\text{п}}$.

Определение 7.5. Автономный процесс P^3, удовлетворяющий ограничениям 1) - 3), назовем замыканием процесса $P^{\text{п}}$.

Пример 7.2. На рисунке 7.2 *а* задан процесс P_1 в структурированных ситуациях которого выделена выходная компонента y^1 (пятая и шестая позиции ситуации), а на рисунке 7.2 б - процесс P_2 с выделенной входной компонентой x^2 (вторая и третья позиции ситуации). Репозиция процесса P_2 показана на рисунке 7.2 в, а автономный процесс, образованный объединением процесса P_2 и его репозиции P_2' - на рисунке 7.2 г. Приведенный процесс, как было сказано выше, получается из автономного процесса исключением путей, ведущих непосредственно в инициатор P_2, т. е. в данном случае на рисунке 7.2 г исключается

дуга, помеченная перекрестием. Если y^1 эквивалентна x^2, то последовательная композиция P_1 и P_2^n будет иметь вид, показанный на рисунке 7.2 д. В ситуациях процесса, образованного композицией, первые четыре позиции - соответствующие позиции процесса P_1, пятая и шестая – компонента $y^1 = x^2$, седьмая и восьмая – первая и вторая позиции ситуаций из P_2, девятая и десятая – пятая и шестая позиции того же процесса.

$\overbrace{011000}^{y_1} \to 011010$ $\overbrace{111000}^{x_1} \to 111010$

а) P_1 б) P_2

$111010 \to 110010 \to 110000 \to 111000$ $111000 \to 111010 \to 110010 \to 110000$

в) P_2' г) p_2^n

$0110\overbrace{00}^{y^1=x^2}1100 \to 0110101100 \to 0110101110$

д) P_3

Рисунок 7.2 – Пример композиции двух процессов

7.3 Согласование асинхронных процессов

Пусть заданы два приведенных процесса $P_1^n = <S_1, F_1, I_1, R_1>$ и $P_2^n = <S_2, F_2, I_2, R_2>$, в ситуациях которых выделены входная и выходная компоненты. Элементы подмножества Y_1^* значений выходной компоненты ситуаций первого процесса ($Y_1^* \subseteq Y_1$) соответствуют (эквивалентны) элементам подмножества X_2^* значений входной компоненты ситуаций второго процесса ($X_2^* \subseteq X_2$), а элементы подмножества Y_2^* значений выходной компоненты ситуаций второго процесса ($Y_2^* \subseteq Y_2$) – элементам подмножества X_1^* значений входной компоненты ситуаций первого процесса ($X_1^* \subseteq X_1$).

Осуществим, если это возможно, последовательную композицию процессов P_1^n и P_2^n считая, что $Y_1^* = X_2^*$ (либо $Y_2^* = X_1^*$). Результатом композиции является асинхронный процесс $P_{1,2}^n(P_{2,1}^n)$. В ситуациях АП $P_{1,2}^n(P_{2,1}^n)$ выделим входную компоненту, совпадающую со входной компонентой ситуаций $P_1^n(P_2^n)$. Построим, если это возможно, замыкание $P_{1,2}^з(P_{2,1}^з)$ АП $P_{1,2}^n(P_{2,1}^n)$, считая, что поэлементно $Y_2^* = X_1^*$ ($Y_1^* = X_2^*$). На основании введенных ранее понятий редукции и замыкания АП доказывается, что $P_{1,2}^з = P_{2,1}^з$.

Ситуации построенного таким образом асинхронного процесса имеют следующую структуру: $s = \{u, v, z\}$, где $u = y^1 = x^2$, $v = y^2 = x^1$, $z = (z^1, z^2)$, причем компоненты u и v принимают значения из множеств $U \subseteq Y_1^* = X_2^*$ и $V \subseteq Y_2^* = X_1^*$ соответственно.

Определение 7.6. Два асинхронных процесса называются согласованными относительно множеств U и V, если результатом применения к ним последовательной композиции и замыкания является автономный процесс, заданный на непустом множестве ситуаций, компоненты u и v которых являются отождествленными входными и выходными компонентами ситуаций этих процессов.

7.4 Формальное определение протокола информационного обмена

Пусть задан асинхронный процесс $P = \langle S, F, I, R \rangle$, в ситуациях которого выделены входная x и выходная y компоненты, определенные на множествах X и Y соответственно. Построим проекции:

1) $S(X \times Y)$ множества S на $X \times Y$;

2) $F(X \times Y)$ множества F на $X \times Y \times X \times Y$;

3) $I(X \times Y)$ множества I на $X \times Y$;

4) $R(X \times Y)$ множества R на $X \times Y$.

Определение 7.7. Асинхронный процесс $P(X \times Y) = \langle S(X \times Y), F(X \times Y), I(X \times Y), R(X \times Y) \rangle$ назовем проекцией процесса P на $X \times Y$.

Определение 7.8. Проекцию автономного процесса $P^{з}_{1,2}$, полученного в результате согласования процессов P^{n}_{1} и P^{n}_{2}, на $U \times V$ будем называть протоколом $П(U, V)$ согласования P^{n}_{1} и P^{n}_{2} относительно множеств U и V.

Протокол $П(U, V)$ является формальным заданием протокола обмена, он реализуется процессом $P^{з}_{1,2}$.

Построим проекции согласованных процессов P^{n}_{1} и P^{n}_{2} на $U \times V$. Очевидно, что эти проекции обладают свойствами:

1) $I_1(U \times V) \subseteq R_2(U \times V)$;

2) $I_2(U \times V) \subseteq R_1(U \times V)$;

3) $F_1(U \times V) \cup F_2(U \times V) = F^{з}_{1,2}(U \times V)$.

Практически задача координации работы двух устройств ставится следующим образом. Имеются два устройства и протокол обмена, которому должно удовлетворять совместное поведение этих устройств. Протокол обмена задает множество отождествляемых значений сигналов (и тем самым входы и выходы, которые нужно соединить друг с другом), а также порядок изменения этих сигналов.

Для решения задачи необходимо:

1) соединяемые устройства описать как АП с выделенными входными и выходными компонентами;

2) определить в соответствии с протоколом обмена множества эквивалентных компонент;

3) согласовать, если это возможно, процессы относительно множеств эквивалентных компонент;

3) построить проекцию АП, описывающего совместное поведение согласованных АП;

4) удостовериться, что полученный протокол согласования описывает заданный протокол обмена (для этого обычно требуется семантическая интерпретация протокола).

Возможны три случая, когда эта задача не решается.

1 Значения выходной компоненты одного АП не эквивалентны значениям входной компоненты другого, например, из-за различия физических параметров сигналов, представляющих эти значения (несогласуемые непосредственно уровни сигналов, их различная физическая природа, различная кодировка сигналов, имеющих один и тот же смысл, и т. д.).

2 Множества U и V, относительно которых АП удалось согласовать, включают не все значения, требуемые протоколом обмена, и из-за этого протокол обмена не реализуется или реализуется не в полном объеме.

3 В автономном процессе, описывающем совместное поведение согласованных процессов, хотя и встречаются все элементы эквивалентных множеств, но не реализуются некоторые последовательности изменений значений компонент, предусмотренные протоколом обмена.

В этих случаях устройства соединяются не непосредственно, а через дополнительный блок, называемый адаптером.

7.5 Согласующий асинхронный процесс

Формально соединение двух устройств через адаптер описывается с помощью реализуемых ими и адаптером трех приведенных АП P_1^n, P_2^n и P_c^n, ситуации которых имеют следующую структуру:

$$s^1 = \{x^1, y^1, z^1), s^2 = \{x^2, y^2, z^2), s^{\mathrm{c}} = \{y^1, y^2, x^1, x^2, z^c).$$

Для ситуаций процесса P_c^n, соответствующего адаптеру, компоненты y^1 и y^2 являются входными, а x^1, x^2 - выходными.

Пара АП P_1^n, P_c^n согласована относительно множеств U_1 и V_1 допустимых значений компонент y^1 и x^1. Процессы P_2^n и P_c^n согласованы относительно множеств U_2 и V_2 допустимых значений компонент y^2 и x^2. Такая композиция АП P_1^n, P_2^n и P_c^n может быть представлена в виде автономного АП $P_{1\mathrm{c}2}$, заданного на ситуациях вида $s^{1\mathrm{c}2} = \{s^1, s^{\mathrm{c}}, s^2)$.

Определение 7.9. Назовем P_c^n асинхронным процессом, согласующим АП P_1^n и P_2^n, если в процессе $P_{1\mathrm{c}2}$:

1) существует такая ситуация s_i^{1c2} с компонентой $s_i' \in R_1$, что любая траектория процесса $P_{1\mathrm{c}2}$ из s_i^{1c2} неизбежно ведет к ситуации s_j^{1c2} с компонентой $s_j^2 \in I_2$;

2) существует такая ситуация s_k^{1c2} с компонентой $s_k^2 \in R_2$, что любая траектория процесса $P_{1\mathrm{c}2}$ из s_k^{1c2} неизбежно ведет к ситуации s_l^{1c2} с компонентой $s_l' \in I_1$.

Иными словами, процесс P_c^n будет согласующим в том случае, если среди его ситуаций найдутся такие, что появление некоторого значения входной компоненты $y^1(y^2)$ по достижении результанта процессом $P_1(P_2)$ с неизбежностью вызывает такое изменение выходной компоненты $x^2(x^1)$, которое приводит в процессе $P_2(P_1)$ к переходу от результанта к инициатору.

Пример 7.3. Рассмотрим систему, состоящую из канального процессора КП, адаптера Ад и двух абонентов - Аб1 и Аб2 (рисунок 7.3 а). КП, Аб1 и Аб2 связаны общей магистралью данных. Каждое из этих устройств имеет возможность передавать информацию в магистраль и принимать ее из магистрали. Система работает по принципу «ведущий - ведомый» и осуществляет обмен информацией между КП и абонентами. Роль ведущего играет КП, а ведомых - абоненты. КП устанавливает адрес A абонента и затем сигнал запроса $a = 1$.

По адресу A и этому сигналу адаптер вырабатывает либо $a_1 = 1$, либо $a_2 = 1$, которые являются сигналами, инициирующими прием или передачу информации. После окончания работы с информацией соответствующий абонент устанавливает сигнал $b_i = 1$. Приняв от абонента $b_i = 1$, адаптер отвечает сигналом $b = 1$. После этого начинается процесс отключения абонента: КП устанавливает $a = 0$ (при этом он может менять адрес), по которому адаптер сбрасывает в 0 тот сигнал a_i, который был равен 1. В ответ на $a_i = 0$ i-й абонент устанавливает $b_i = 0$, после чего адаптер выставляет $b = 0$, и система оказывается в исходном состоянии.

На рисунке 7.3 б изображена сеть Петри, задающая взаимодействие указанных устройств на уровне установления и разрыва связи. Эта сеть Петри описывает согласованный АП и состоит из четырех фрагментов, относящихся к перечисленным выше устройствам. В силу того, что КП может обратиться только к одному из абонентов, точка может появиться либо в кружке 5, либо в кружке 6, а значит, либо в 7, либо в 8, но не в обоих одновременно.

Для выделенных фрагментов перечислим ситуации, являющиеся инициаторами и результантами.

Результанты фрагмента КП:

1) точки в кружках 4 и 7 - установка адреса Аб1 и сигнала $a = 1$;

2) точки в кружках 4 и 8 - установка адреса Аб2 и сигнала $a = 1$;

3) точки в кружках 2 и 4 - установка $a = 0$.

Инициатором фрагмента КП является результант 1) фрагмента Ад.

Результанты фрагмента Ад:

1) точка в кружке 9 - установка $b = 1$ или $b = 0$;

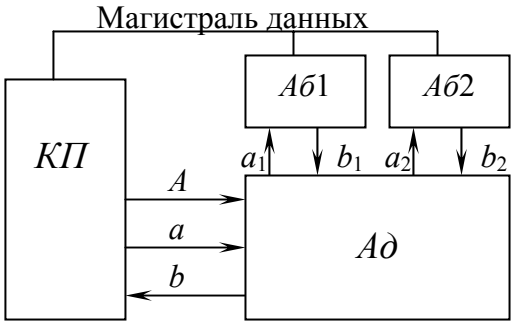
2) точка в кружке 11 - установка $a_1 = 1$ или $a_1 = 0$;

3) точка в кружке 13 - установка $a_2 = 1$ или $a_2 = 0$.

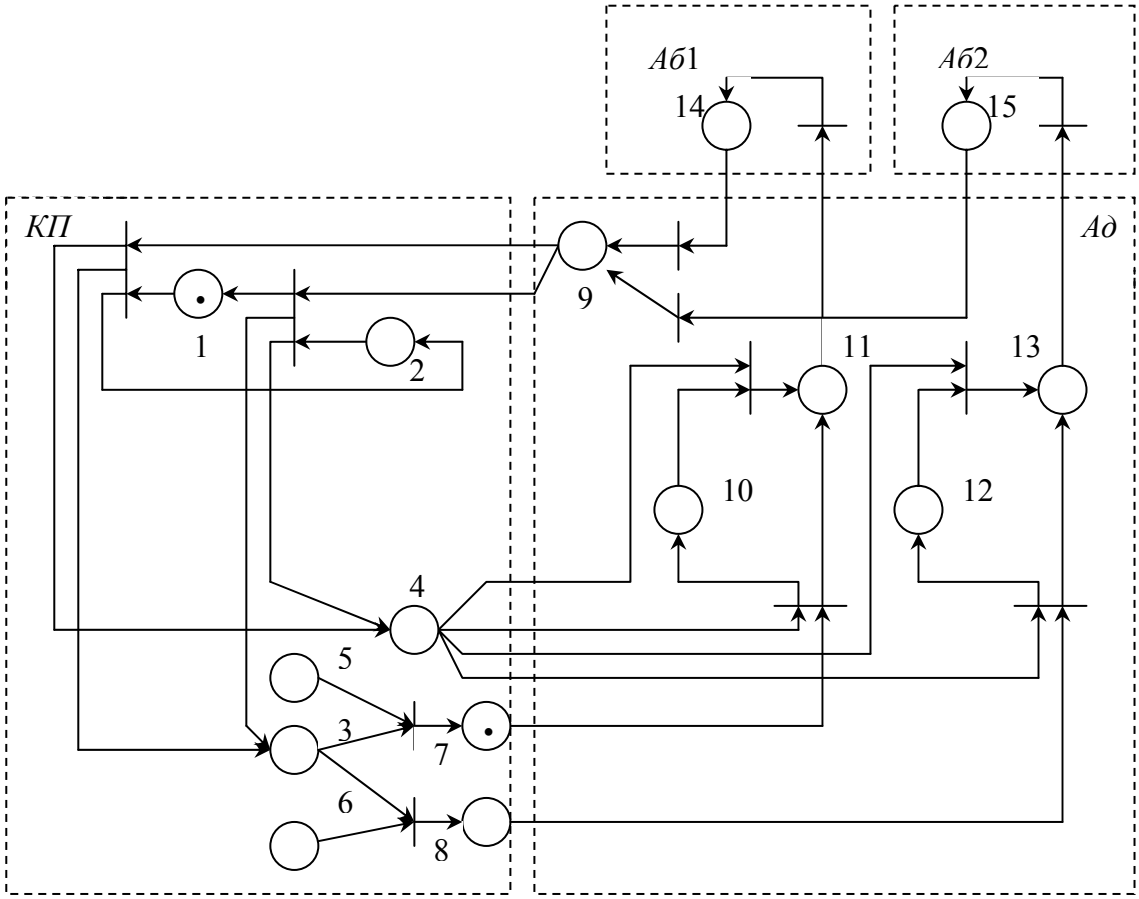
Инициаторы фрагмента Ад - результанты фрагментов КП, Аб1 и Аб2.

Фрагмент Аб1: результант - точка в кружке 14 (установка $b_1 = 1$ или $b_1 = 0$), инициатор - результант 2) фрагмента Ад.

Фрагмент Аб2: результант - точка в кружке 15 (установка $b_2 = 1$ или $b_2 = 0$), инициатор - результант 3) фрагмента Ад.



а) структура



б) протокол установления и разрыва связей на языке сетей Петри

Рисунок 7.3 - Система связи канального процессора (КП) с двумя абонентами (Аб1, Аб2) через адаптер (Ад)

Замечание. В приведенном примере уровень протокола, заданного на рисунке 7.3 б таков, что он непосредственно может быть использован для реализации адаптера. Это связано с тем, что данная сеть относится к классу устойчивых и безопасных сетей Петри, для которых разработаны методы синтеза.

Результатом синтеза является управляющий автомат адаптера, композиция которого с операционной частью является реализацией адаптера.

Вопросы и задания для самоконтроля

1 Определите понятие интерфейс в широком смысле.

2 Охарактеризуйте уровни интерфейса.

3 Что является предметом теории протоколов?

4 Каким требованиям должен удовлетворять протокол?

5 Перечислите этапы разработки интерфейсного взаимодействия между устройствами.

6 Для асинхронного процесса, заданного диаграммой переходов на рисунке 7.4, постройте:

а) репозицию процесса P';

б) приведенный асинхронный процесс $P^{\text{п}}$;

в) редукцию асинхронного процесса $P(X^*)$ по множеству $X^*=\{00, 01\}$.

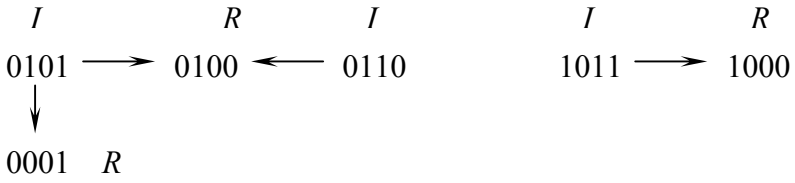


Рисунок 7.4 – Диаграмма переходов процесса P

7 Постройте последовательную композицию редуцированных асинхронных процессов P_1 и P_2^n, представленных на рисунке 7.5. В качестве выходной компоненты процесса P_1 взять вторые два элемента вектора полного состояния, а в качестве входной компоненты процесса P_2 – второй и третий элемент вектора полного состояния.

Рисунок 7.5 – Редуцированные асинхронные процессы

8 Каким требования должен удовлетворять автономный процесс, являющийся замыканием приведенного АП?

9 Какие два асинхронных процесса называются согласованными?

10 Дайте формальное определение протокола информационного обмена.

11 Перечислите этапы решения задачи координации работы двух устройств.

12 Назовите случаи, когда задача координации работы устройств не будет иметь решения.

13 Как формально описывается соединение двух устройств через адаптер?

Тесты проверки усвоения материала

1 Логический уровень интерфейса определяет требования к:
а) линиям связи;
б) разъемным соединителям;
в) способу представления сигналов, символов и сообщений;
г) токовой нагрузке на источник питания;
д) допускам на параметры электрических сигналов.

2 На этапе абстрактного синтеза протокола выполняется:
а) выбор формализованного языка описания протокола;
б) кодирование состояний и сигналов системы;
в) построение реализаций устройства в элементной базе;
г) описание соединяемых устройств как асинхронных процессов;
д) согласование асинхронных процессов.

3 В репозиции асинхронного процесса $P = <S, F, I, R>$ множество ситуаций S' определяется как:
а) $S' \subseteq I \cup R \cup S^{д}$;
б) $S' \subseteq I \cup R$;
в) $S' \subset I \cup R \cup S^{д}$;
г) $S' \subseteq I \cup S^{д}$;
д) $S' = I \cup R \cup S^{д}$.

4 В репозиции асинхронного процесса $P = <S, F, I, R>$ множество инициаторов определяется как:
а) $I' = R$;
б) $I' \subset R$;
в) $I' \subseteq I$;
г) $I' \subset I$;
д) $I' \subseteq R$.

5 Приведенный асинхронным процесс $P^{п}$ является объединением асинхронного процесса P и его репозиции P', для которого справедливо:
а) $F^{п} = F \cap (F' \setminus (F' \cap (S' \times I)))$;
б) $F^{п} = F \cap (F' \setminus (F' \cup (S' \times I)))$;
в) $F^{п} = F \cup (F' \setminus (F' \cup (S' \times I)))$;
г) $F^{п} = F \cup (F' \setminus (F' \cap (S' \times I)))$;
д) $F^{п} = F' \cup (F \setminus (F' \cup (S \times I)))$.

6 В приведенном асинхронном процессе $P^{п}$, построенном для процесса P, инициаторы и результанты удовлетворяют условию:
а) $I^{п} \subseteq I$, $R^{п} = R \cup S^{д}$;
б) $I^{п} \subseteq I \cup S^{д}$, $R^{п} = R \cup S^{д}$;

в) $I^{\text{п}} \subseteq I \cup S^{\text{д}}$, $R^{\text{п}} = R$;
г) $I^{\text{п}} \subseteq R$, $R^{\text{п}} = I \cup S^{\text{д}}$;
д) $I^{\text{п}} \subseteq R \cup S^{\text{д}}$, $R^{\text{п}} = I \cup S^{\text{д}}$.

7 Для редукции процесса, представленного на рисунке 7.4, по последним двум элементам вектора состояния, справедливо:
а) $S(01, 00) = \{0101, 0100, 0110, 0001, 1011, 1000\}$;
б) $S(01, 00) = \{0101, 0100, 0110, 0001\}$;
в) $S(01, 00) = \{0101, 0100, 0001, 1000\}$;
г) $S(01, 00) = \{0001, 1011, 1001, 1000\}$;
д) $S(01, 00) = \{0101, 0100, 0110, 0001, 1011\}$.

8 Для редукции процесса, представленного на рисунке 7.4, по последним двум элементам вектора состояния, справедливо:
а) $I(01, 11) = \{0101, 1011, 0110\}$;
б) $I(01, 11) = \{1011, 0110\}$;
в) $I(01, 11) = \{0101, 0110\}$;
г) $I(01, 11) = \{0101, 1011\}$;
д) $I(01, 11) = \{0101\}$.

9 Для редукции процесса, представленного на рисунке 7.4, по последним двум элементам вектора состояния, справедливо:
а) $R(00) = \{0100, 1000, 0001\}$;
б) $R(00) = \{0100, 0001\}$;
в) $R(00) = \{1000, 0001\}$;
г) $R(00) = \{1000\}$;
д) $R(00) = \{0100, 1000\}$.

10 Для последовательной композиции P_3 асинхронных процессов P_1 и P_2^n, образованной отождествлением входной X_2 и выходной Y_1 компонент, справедливо:
а) $S_3 \subseteq S_1(Y_1^*) \cup S_2(X_2^*)$;
б) $S_3 \subseteq S_1(Y_1^*) \times S_2(X_2^*)$;
в) $S_3 \subseteq S_1(Y_1^*) \cap S_2(X_2^*)$;
г) $S_3 = S_1(Y_1^*) \cap S_2(X_2^*)$;
д) $S_3 \subset S_1(Y_1^*) \times S_2(X_2^*)$.

11 Для последовательной композиции P_3 асинхронных процессов P_1 и P_2^n, образованной отождествлением входной X_2 и выходной Y_1 компонент, справедливо:
а) если в $s^3(s^1, s^2)$ компонента $s^2 \in I_2(X_2^*)$, то $s^1 \in R_1(Y_1^*)$;
б) если в $s^3(s^1, s^2)$ компонента $s^1 \in I_2(X_2^*)$, то $s^2 \in R_1(Y_1^*)$;
в) если в $s^3(s^1, s^2)$ компонента $s^2 \in R_2(X_2^*)$, то $s^1 \in I_1(Y_1^*)$;

г) если в $s^3(s^1, s^2)$ компонента $s^2 \subset I_2(X_2^*)$, то $s^1 \subset R_1(Y_1^*)$;

д) если в $s^3(s^1, s^2)$ компонента $s^1 \subset I_2(X_2^*)$, то $s^2 \subset R_1(Y_1^*)$.

12 Для последовательной композиции асинхронных процессов P_1 и P_2^n, заданных на рисунке 7.5, справедливо:

а) 001001→000101;

б) 000101→ 001001;

в) 101000→ 001001;

г) 010001→ 100001;

д) 000101→ 000110.

13 Для замыкания процесса $P^з$, построенного по приведенному процессу $P^п$ и его репозиции P', при выделенных входных X и выходных Y компонентах, верно:

а) $S^3 \subseteq S^п(X^*) \times S^п(Y^*)$;

б) $S^3 \subseteq S^п(X^*) \cup S^п(Y^*)$;

в) $S^3 \subseteq S^п(X^*) \cap S^п(Y^*)$;

г) $S^3 = S^п(X^*) \cap S^п(Y^*)$;

д) $S^3 = S^п(X^*) \cup S^п(Y^*)$.

14 Для замыкания процесса $P^з$, построенного по приведенному процессу $P^п$ и его репозиции P', при выделенных входных X и выходных Y компонентах, верно:

а) $y^3=x^3$;

б) $y^3=x^п$;

в) $y^3 = x'$;

г) $x^3=y^п$;

д) $x^3=y'$.

15 Для замыкания процесса $P^з$, построенного по приведенному процессу $P^п$ и его репозиции P', при выделенных входных X и выходных Y компонентах, верно:

а) $F^3 \subset (F(X^*) \cup F'(X^*)) \cap (F'(Y^*) \cup F(Y))$;

б) $F^3 = (F(X^*) \cup F'(X^*)) \times (F'(Y^*) \cup F(Y))$;

в) $F^3 = (F(X^*) \cap F'(X^*)) \cup (F'(Y^*) \cap F(Y))$;

г) $F^3 \subset (F(X^*) \cap F'(X^*)) \cup (F'(Y^*) \cap F(Y))$;

д) $F^3 = (F(X^*) \cup F'(X^*)) \cap (F'(Y^*) \cup F(Y))$.

16 Протоколом согласования приведенных процессов относительно множеств U и V называют:

а) замыкание последовательной композиции приведенных процессов на $U \times V$;

б) проекцию замыкания последовательной композиции приведенных процессов на $U \times V$;

в) редукцию замыкания последовательной композиции приведенных процессов на $U \times V$;

г) замыкание проекции последовательной композиции приведенных процессов на $U \times V$;

д) последовательную композицию проекции замыкания приведенных процессов на $U \times V$.

17 В случае, если задача координации работы устройств не решается, устройства соединяются через:

а) коннектор;

б) адаптер;

в) интерфейс;

г) репитер;

д) трансивер.

18 Формально соединение двух устройств через адаптер описывается с помощью:

а) трех редуцированных процессов, реализуемых устройствами и адаптером;

б) трех приведенных процессов, реализуемых устройствами;

в) трех репозиций процессов, реализуемых устройствами и адаптером;

г) последовательной композиции трех приведенных процессов, реализуемых устройствами и адаптером;

д) замыкания трех приведенных процессов, реализуемых устройствами и адаптером.

19 Если ситуации процессов, описывающих соединяемые устройства, имеют вид $s^1 = \{x^1, y^1, z^1\}$, $s^2 = \{x^2, y^2, z^2\}$, то для процесса адаптера:

а) $s^c = \{y^1, y^2, z^c, x^1, x^2\}$;

б) $s^c = \{x^1, x^2, y^1, y^2, z^c\}$;

в) $s^c = \{x^1, y^1, x^2, y^2, z^c\}$;

г) $s^c = \{y^1, y^2, x^1, x^2, z^c\}$;

д) $s^c = \{y^1, x^1, x^2, y^2, z^c\}$.

20 Если ситуации процессов, описывающих соединяемые устройства, имеют вид $s^1 = \{x^1, y^1, z^1\}$, $s^2 = \{x^2, y^2, z^2\}$, то для процесса адаптера входными будут компоненты:

а) y^1, y^2;

б) y^1;

в) x^1, x^2;

г) x^1;

д) z^1, z^2.

8 Итоговые тесты проверки усвоения материала

1 Тестом в схеме программы является слово:

а) $p(a) > g(h(x, y))$;

б) $p(1, g(h(x, y)))$;

в) $p(a, g(h(x, 3)))$;

г) $f(a, g(h(x, y))) = 0$;

д) $f(a) + g(h(x, y)) = 0$.

2 Семантика, опирающаяся на теорию рекурсивных функций, называется:

а) операционной;

б) аксиоматической;

в) денатационной;

г) декларативной;

д) императивной.

3 Множество результантов R асинхронного процесса удовлетворяет условию:

а) если sFr, $r \in R$, то $s \in R$;

б) если rFs, $r \in R$, то $s \notin R$;

в) если rFs, $r \in R$, то $s \in R$;

г) если sFr, $r \in R$, то $s \notin R$;

д) если sMr, $r \in R$, то $s \in R$.

4 В сети Петри, представленной на рисунке 8.1, могут сработать переходы:

а) t_1 и t_3;

б) t_1 и t_2;

в) t_2 и t_3;

г) t_2;

д) t_3.

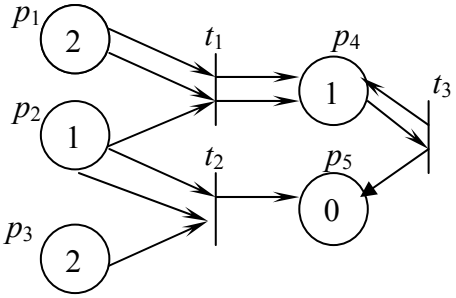


Рисунок 8.1 – Граф сети Петри

5 Вычислительные процессы, множества переменных которых не пересекаются, называются:

а) независимыми;

б) взаимоисключающими;

в) асинхронными;

г) сотрудничающими;

д) конкурирующими.

6 Допустимая последовательность исполнения вычислительной схемы, представленной на рисунке 8.2:

а) S_1, S_2, S_4, S_3, S_5, S_1;

б) S_1, S_4, S_5, S_1, S_2, S_3;

в) S_1, S_2, S_3, S_4, S_5, S_1;

г) S_2, S_3, S_1, S_4, S_5, S_1;

д) S_1, S_4, S_2, S_3, S_5, S_1.

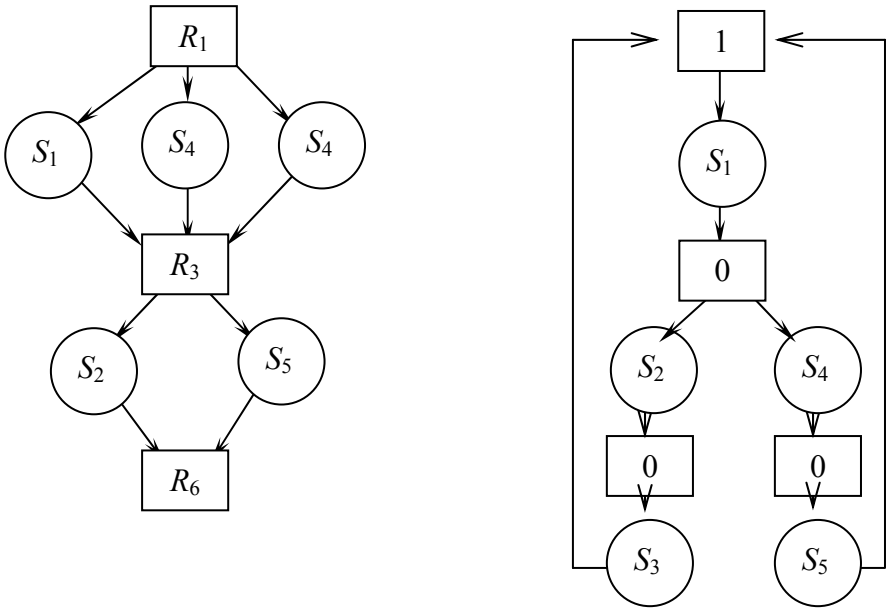


Рисунок 8.2– Вычислительная схема процесса

7 Совокупность правил, обеспечивающих совместное функционирование аппаратных и программных средств, называют:

а) редукцией;

б) адаптером;

в) интерфейсом;

г) протоколом;

д) репозицией.

8 Функциональные выражения в схемах программ могут содержать:

а) +, - , *, /;

б) $\wedge$, $\vee$, $\neg$;

в) *cos*, *sin*, *tg*;

г) >, <, =, <=, >=;

д) 0, 1, 2, 3, …

9 Денотационное отображение для синтаксического правила <восьмеричное_число> ::= <восьмеричное_число> 3 имеет вид:

а) M_o(<восьмеричное_число> '3') = 3*M_o(<восьмеричное_число>) + 8;

б) M_o(<восьмеричное_число> '3') = 3*M_o(<восьмеричное_число>);

в) M_o(<восьмеричное_число> '3') = M_o(<восьмеричное_число>) + 3;

г) M_o(<восьмеричное_число> '3') = 8*M_o(<восьмеричное_число>) + 3;

д) M_o(<восьмеричное_число> '3') = M_o^8(<восьмеричное_число>) + 3.

10 Множество инициаторов I эффективного асинхронного процесса должно удовлетворять условию:

а) $(\forall s \in S \backslash I)\ (\exists\ i \in R)$: sMi;

б) $(\forall s \in S \backslash I)\ (\exists\ i \in I)$: iMs;

в) $(\forall i \in I)\ (\exists\ s \in S \backslash I)$: iMs;

г) $(\forall i \in I)\ (\exists\ s \in S)$: sMi;

д) $(\forall s \in S)\ (\exists\ i \in I)$: iMs.

11 Для сети Петри, представленной на рисунке 8.1, тупиковой будет разметка:

а) 00010;

б) 12100;

в) 21000;

г) 22200;

д) 30303.

12 Переменная специального типа, доступная параллельным процессам для проведения над ней только двух операций – «закрытия» и «открытия»:

а) почтовый ящик;

б) монитор;

в) очередь;

г) семафор;

д) конвейер.

13 В вычислительной схеме, представленной на рисунке 8.2, пятый оператор соперничает за регистр с операторами:

а) 1 и 3;

б) 4;

в) 1, 2, 3 и 4;

г) 3;

д) 1, 2 и 3.

14 Координацию протекания информационных потоков между устройствами обеспечивает:

а) механический уровень интерфейса;

б) электрический уровень интерфейса;

в) механический и электрический уровень интерфейса;

г) логический уровень интерфейса;

д) электрический и логический уровень интерфейса.

15 Оператором засылки в схемах программ является:

а) $x:=b$;

б) $x:=x+5$;

в) $x:=x+y$;

г) $x:=5$;

д) $x:=z$.

16 Предикат, описывающий постусловие программы *Pg* в аксиоматической семантике на множестве состояний памяти *State*:

а) *post-Pg*: $State \rightarrow Boolean$;

б) *post-Pg*: $Boolean \rightarrow State \times State$;

в) *post-Pg*: $State \times State \rightarrow Boolean$;

г) *post-Pg*: $Boolean \rightarrow State$;

д) *post-Pg*: $Boolean \times Boolean \rightarrow State$.

17 Для множества инициаторов *I* простого асинхронного процесса выполняется условие:

а) $(\forall s \in S \backslash I)\ (\forall i \in I)\ sFi \Rightarrow s \in I$;

б) $(\forall s \in S)\ (\forall i \in I)\ sFi \Rightarrow i \in I$;

в) $(\forall s \in S \backslash I)\ (\forall i \in I)\ sFi \Rightarrow s \notin I$;

г) $(\forall s \in S)\ (\forall i \in I)\ sFi \Rightarrow s \notin I$;

д) $(\forall s \in S)\ (\forall i \in I)\ iFs \Rightarrow s \notin I$.

18 В сети Петри, представленной на рисунке 8.1, при срабатывании перехода t_1 новая разметка будет:

а) 01220;

б) 00240;

в) 00230;

г) 10210;

д) 01030.

19 Допустимое значение мьютекса:

а) -11;

б) 5.8;

в) 1;

г) 2;
д) 0.3E-1.

20 Для модели пространства состояний системы, представленной на рисунке 8.3, справедливо:

а) $S_3 \xrightarrow{*} S_6$;

б) $S_1 \xrightarrow{*} S_4$;

в) $S_5 \xrightarrow{*} S_2$;

г) $S_4 \xrightarrow{*} S_5$;

д) $S_3 \xrightarrow{*} S_5$.

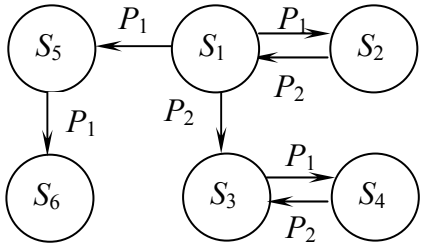


Рисунок 8.3 – Модель пространства состояний системы

21 Оъединение асинхронного процесса и его репозиции, в котором из отношения F выброшены пары, задающие переходы к инициаторам, определяет:

а) приведенный процесс;
б) редукцию процесса;
в) замыкание процесса;
г) проекцию процесса;
д) последовательную композицию процессов.

22 Конечное множество переменных схемы S составляют ее:

а) протокол;
б) интерпретацию;
в) детерминант;
г) конфигурацию;
д) память.

23 Для описания семантики отдельного оператора программы в аксиоматической семантике используются:

а) инварианты;
б) тройки Хоара;
в) аксиомы;
г) правила вывода;
д) ограничивающие функции.

24 Асинхронный процесс, граф которого представлен на рисунке 8.4, простой при:

а) $I = \{S_1, S_2\}$; $R = \{S_5, S_6\}$;

б) $I = \{S_1, S_3\}$; $R = \{S_5, S_6\}$;

в) $I = \{S_1, S_3\}$; $R = \{S_4, S_5\}$;

г) $I = \{S_1, S_3\}$; $R = \{S_4, S_5, S_6\}$;

д) $I = \{S_1\}$; $R = \{S_5, S_6\}$.

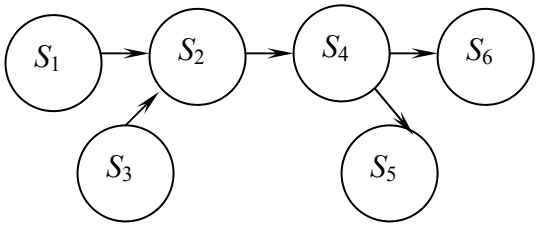


Рисунок 8.4 – Граф асинхронного процесса

25 В сети Петри, представленной на рисунке 8.1, переходы t_1 и t_2 будут совместно возможными при разметке:

а) 21210;

б) 21111;

в) 13110;

г) 23200;

д) 23010.

26 При реализации операции открытия семафора $V(S)$, если $S<0$, то будет выполнена команда:

а) *WAIT*(*S*);

б) *RELEASE*(*S*);

в) *S*:=*S*-1;

г) *CS* {критическая секция};

д) *PR* {процесс вне критического интервала}.

27 Для модели пространства состояний системы, представленной на рисунке 8.3, справедливо:

а) $P_2(S_1) = \{S_2\}$;

б) $P_1(S_6) = \{S_5\}$;

в) $P_1(S_5) = \{S_6\}$;

г) $P_2(S_3) = \{S_4\}$;

д) $P_2(S_5) = \{S_6\}$.

28 Для редукции процесса, представленного на рисунке 8.5, по первым двум элементам вектора состояния, справедливо:

а) *S*(01, 00) = {0101, 0100, 0110, 0001, 1011, 1000};

б) *S*(01, 00) = {0101, 0100, 0110, 0001};

в) $S(01, 00) = \{0100, 0101, 0001\}$;
г) $S(01, 00) = \{0001, 1011, 1001, 1000\}$;
д) $S(01, 00) = \{0101, 0100, 0110, 0001, 1011\}$.

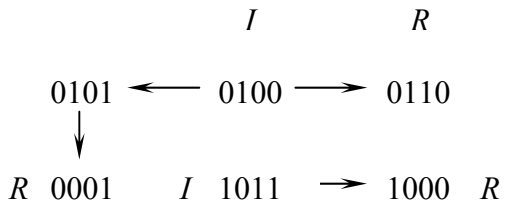


Рисунок 8.5 – Асинхронный процесс

29 Если в схемах программ $\tau = a$, a – константа, то $\tau_I(W) =$
а) $I(a)$;
б) $W(a)$;
в) $\tau_I(a)$;
г) $W_I(\tau)$;
д) a.

30 Истинная тройка Хоара для оператора цикла:
а) $\{x<0\}$ *while* $x<1$ *do* $x:=x+1$; $\{x>1\}$;
б) $\{x<0\}$ *while* $x<1$ *do* $x:=x+1$; $\{x<1\}$;
в) $\{x<0\}$ *while* $x<1$ *do* $x:=x+1$; $\{x=0\}$;
г) $\{x<0\}$ *while* $x<1$ *do* $x:=x+1$; $\{x=1\}$;
д) $\{x<0\}$ *while* $x<1$ *do* $x:=x+1$; $\{x<0\}$.

31 Вычислительный процесс, граф которого представлен на рисунке 8.4, при $I = \{S_1, S_3\}$ и $R = \{S_4, S_5, S_6\}$ является:

а) синхронным;

б) эффективным;

в) управляемым;

г) простым;

д) автономным.

32 Каждая безопасная сеть Петри является:

а) ограниченной;

б) живой;

в) сохраняющей;

г) покрываемой;

д) устойчивой.

33 При реализации операции закрытия семафора $P(S)$, если $S<0$, то будет выполнена команда:

а) *WAIT*(*S*);
б) *RELEASE*(*S*);
в) *S*:=*S*+1;
г) *CS* {критическая секция};
д) *PR* {процесс вне критического интервала}.

34 В модели пространства состояний системы, представленной на рисунке 8.3, безопасными являются состояния:
а) 1 и 2;
б) 5 и 6;
в) 5;
г) 3 и 4;
д) 1.

35 Для редукции процесса, представленного на рисунке 8.5, по последним двум элементам вектора состояния, справедливо:
а) $I(01, 11) = \{0101, 1011, 0110\}$;
б) $I(01, 11) = \{1011, 0110\}$;
в) $I(01, 11) = \{0100, 1011\}$;
г) $I(01, 11) = \{1011\}$;
д) $I(01, 11) = \{0100\}$.

36 Если в протоколе выполнения программы $l_{i+1} = k$, $W_{i+1} \overset{x}{=} W_i$, то оператор схемы в вершине с меткой l_i является:
а) начальным оператором;
б) условным оператором;
в) оператором присваивания;
г) оператором петли;
д) конечным оператором.

37 Закон консеквенции, задающий ослабление постусловия тройки Хоара имеет вид:
а) $P \to R, \{R\}A\{Q\} \vdash \{P\}A\{R\}$;
б) $P \to R, \{R\}A\{Q\} \vdash \{P\}A\{Q\}$;
в) $\{P\}A\{R\}, R \to Q \vdash \{R\}A\{Q\}$;
г) $\{P\}A\{R\}, R \to Q \vdash \{P\}A\{R\}$;
д) $\{P\}A\{R\}, R \to Q \vdash \{P\}A\{Q\}$.

38 В диаграмме переходов, представленной на рисунке 8.6, конфликтной является ситуация:
а) S_2;
б) S_4;
в) S_3;

г) S_5;

д) S_1.

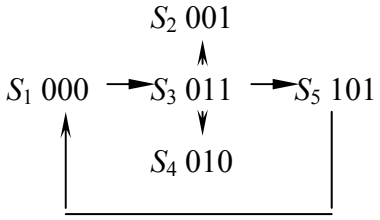


Рисунок 8.6 – Диаграмма переходов

39 Если при любой разметке $M' \in R(N, M)$ переход t может сработать, то он является:

а) потенциально мертвым при разметке M;

б) живым в сети N;

в) мертвым в сети N;

г) потенциально живым при разметке M;

д) устойчивым при разметке M.

40 Средство обмена данными между процессами, представляющее собой реализацию очереди на массивах:

а) *queue*;

б) *monitor*;

в) *semafor;*

г) *pipe*;

д) *TS*.

41 В модели пространства состояний системы, представленной на рисунке 8.3, состояния 5 и 6 являются:

а) тупиковыми;

б) опасными;

в) безопасными;

г) фиксированными;

д) устойчивыми.

42 Для редукции процесса, представленного на рисунке 8.5, по второму и третьему элементу вектора состояния, справедливо:

а) $R(10, 11) = \{0110, 1000, 0001\}$;

б) $R(10, 11) = \{0110, 0001\}$;

в) $R(10, 11) = \{1000, 0001\}$;

г) $R(10, 11) = \{0110\}$;

д) $R(10, 11) = \{1000\}$.

43 Если вершина схемы программы с меткой l_i помечена условным оператором, а выходящая из нее Δ-дуга ведет к вершине с меткой k, то в протоколе:

а) $l_{i+1} = k$ и $W_{i+1} = W_i$;

б) $l_{i+1} = k$, $W_{i+1} \overset{x}{=} W_i$;

в) $l_{i+1} = l_k$, $W_{i+1} \overset{x}{=} W_i$;

г) $l_{i+1} = l_i$ и $W_{i+1} = W_i$;

д) $l_{i+1} = l_k$ и $W_{i+1} = W_i$.

44 Для оператора $\{x^4 = 16\}$ $x{:=}x*x$ $\{Q\}$ истинным постусловием будет:

а) $x = 4$;

б) $x = 2$;

в) $x = 8$;

г) $x{=}16$;

д) $x = 32$.

45 В диаграмме переходов, представленной на рисунке 8.6, все компоненты возбуждены в ситуации:

а) S_2;

б) S_4;

в) S_5;

г) S_3;

д) S_1.

46 Переход t_1 сети Петри, покрывающее дерево которой представлено на рисунке 8.7, является:

а) живым переходом сети;

б) потенциально мертвым при M_2;

в) потенциально живым при M_3;

г) мертвым переходом сети;

д) потенциально мертвым при M_1.

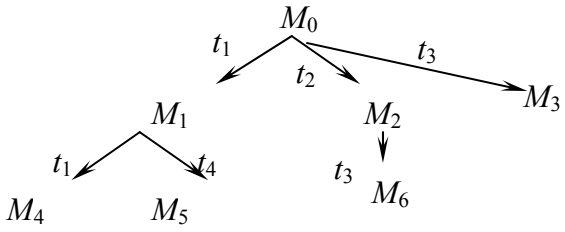


Рисунок 8.7 – Покрывающее дерево сети Петри

47 Удаление из очереди всех сообщений реализует команда:

а) *ReadQueue*;

б) *WriteQueue*;

в) *PurgeQueue*;

г) *QueryQueue*;

д) *PeekQueue*.

48 В модели Холта, представленной на рисунке 8.8, в текущий момент может редуцировать процесс:

а) 1;
б) 2;
в) 1 и 3;
г) 2 и 3;
д) 3.

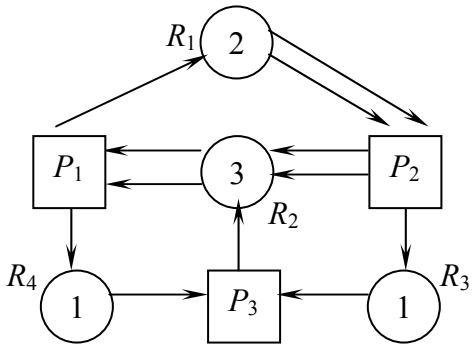


Рисунок 8.8 – Модель Холта системы

49 Для последовательной композиции асинхронных процессов P_1 и P_2^n, заданных на рисунке 8.9, справедливо:
а) 001001→000101;
б) 000111→ 001001;
в) 101000→ 001001;
г) 010001→ 100001;
д) 000101→ 000110.

Рисунок 8.9– Редуцированные процессы

50 Если для любой свободной интерпретации I_h программа (S, I_h) останавливается, то схема S в базисе ß:
а) пустая;
б) правильная;
в) стандартная;
г) тотальная;
д) полная.

51 При заданном предусловии P и постусловии Q оператора цикла, представленного на рисунке 8.10, точке 2 будет соответствовать условие:
а) P;
б) $P \wedge B$;
в) Q;
г) $P \wedge \overline{B}$;
д) $Q \wedge B$.

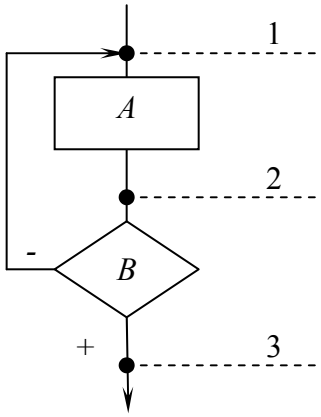


Рисунок 8.10 – Схема оператора цикла

52 Переходу $S_3 – S_4$ диаграммы на рисунке 8.6 в маркированном графе соответствует вершина:

а) z_2^-;
б) z_3^+;
в) z_1^+;
г) z_2^+;
д) z_3^-.

53 Потенциально мертвыми при разметке M_2 сети Петри, покрывающее дерево которой представлено на рисунке 8.7, являются переходы:

а) t_1 и t_3;
б) t_1, t_2 и t_3;
в) t_3 и t_4;
г) t_1, t_2 и t_4;
д) t_2 и t_4.

54 Дисциплина обслуживания сообщений конвейером по принципу *FIFO* предусматривает то, что:

а) сообщение, записанное последним, будет прочитано первым;
б) сообщение, записанное последним, будет прочитано первым или последним;
в) сообщение будет прочитано в соответствии с его приоритетом;
г) сообщение будет прочитано в произвольном порядке;
д) сообщение, записанное первым, будет первым прочитано.

55 В модели Холта, представленной на рисунке 8.8, текущее состояние является тупиком для процессов:

а) 1;
б) 1 и 2;
в) 2;
г) 1 и 3;
д) 2 и 3.

56 Если ситуации процессов, описывающих соединяемые устройства, имеют вид $s^1 = \{x^1, y^1, z^1\}$, $s^2 = \{x^2, y^2, z^2\}$, то для процесса адаптера выходными будут компоненты:

а) y^1, y^2;
б) y^1;
в) x^1, x^2;
г) x^1;
д) z^1, z^2.

57 Если в ДГКА входной алфавит $T=\{a, b, c, d\}$, то в ДДГКА символу c будет соответствовать код:
а) 0000;
б) 1111;
в) 1110;
г) 1000;
д) 0001.

58 При заданном предусловии P и постусловии Q оператора цикла, представленного на рисунке 8.10, условие $Q \wedge B$ будет соответствовать точке:
а) 1;
б) 3;
в) 2;
г) 1 и 3;
д) 1 и 2.

59 Свободная интерпретация I такая, что $I(p)('f^i\alpha') = \begin{cases} 1, & i = 2^n - n, 1 \le n \le 3; \\ 0, & \textit{иначе.} \end{cases}$ согласована со словом α вида:
а) 11001;
б) 10101;
в) 01010;
г) 10011;
д) 11100.

60 Для пути β, заданного на рисунке 8.11, условие пути будет иметь вид:

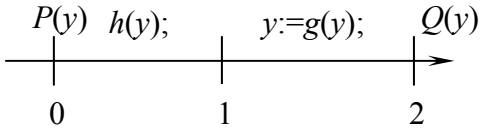


Рисунок 8.11 – Путь β

а) $(h(g(y)) \to Q(g(y))) \to P(y)$;
б) $P(y) \to (Q(g(y)) \to h(g(y)))$;
в) $(P(y) \to h(g(y))) \to Q(g(y))$;
г) $P(y) \to h(g(y)) \wedge Q(g(y))$;
д) $P(y) \to (h(g(y)) \to Q(g(y)))$.

Список использованных источников

1 Рабинович, Е. В. Теория вычислительных процессов [Текст]: учеб. пособие / Е. В. Рабинович; М-во РФ по связи и информатизации, Гос. образоват. учреждение высш. проф. образования «Сиб. гос. ун-т телекоммуникаций и информатики». - Новосибирск: СибГУТИ, 2004. - 119 с.: ил.; 20 см. - Библиогр.: с. 118. – 100 экз.

2 Котов, В. Е. Теория схем программ [Текст] / В. Е. Котов. - М.: Наука, 1991 – 247, [1] с.: ил.; 22 см. - Библиогр.: с. 239-245. – 7250 экз. - ISBN 5-02-013974-2.

3 Калинин, А. Г. Универсальные языки программирования: Семантический подход [Текс] / А. Г. Калинин, И. В. Мацкевич. – М.: Радио и связь, 1991. - 398, [1] с.: ил.; 21 см. - Библиогр.: с. 395-398. – 10000 экз. – ISBN 5-256-00638-X (в пер.).

4 Семантика языков программирования [Текст]: Сборник статей / Пер. с англ. А. Н. Бирюкова, В. А. Серебрякова; под общ. ред. В. М. Курочкина. – М.: Мир, 1980. – 395 с.; 22 см. - Библиогр. в конце статей. – 8000 экз.

5 Прохорова, О. В. Теория вычислительных процессов и структур [Текст]: Учеб. пособие для студентов по спец. 654400 «Телекоммуникации» / О. В. Прохорова; М-во образования РФ, Мар. гос. техн. ун-т. - Йошкар-Ола: МарГТУ, 2001. - 106 с.: ил.; 20 см. - Библиогр.: с. 106. – 200 экз. – ISBN 5-8158-0133-X.

6 Зайдуллин, С. С. Теория вычислительных процессов [Текст]: учеб. пособие / С. С. Зайдуллин, В. Н. Яхонтов; акад. упр. «ТИСБИ». - Казань: ТИСБИ, 2005. - 136 с.: ил.; 20 см. - Библиогр.: с. 135-136. – 300 экз. – ISBN 5-93593-073-0 (в обл).

7 Кораблин, Ю. П. Семантика языков программирования [Текст]: учеб. пособие по курсу «Семантика языков программирования» / Ю. П. Кораблин; под общ. ред. В. П. Кутепова; М-во науки, высш. шк. и техн. политики Рос. Федерации, Моск. энерг. ин-т. - М.: Изд-во МЭИ, 1992. - 100 с.; 21 см. - Библиогр.: с. 99. – 500 экз.

8 Соколов, А. П. Системы программирования: теория, методы, алгоритмы [Текст]: учеб. пособие для студентов, обучающихся по направлению 654600 - Информатика и вычисл. техника / А. П. Соколов. – М.: Финансы и статистика, 2004. – 319, [1] с.: ил.; 21 см. - Библиогр.: с. 309-310. – Предм. указ.: с. 313-320. – 4000 экз. – ISBN 5-279-02770-7.

9 Варшавский, В. И. Автоматное управление асинхронными процессами в ЭВМ и дискретных системах [Текст] / В. И. Варшавский, М. А. Кишиневский, В. Б. Мараховский; под. общ. ред. В.И. Варшавского. - М.: Наука, 1986. – 398 с.: ил.; 21 см. – Авт. указаны на обороте тит. л. - Библиогр.: с. 374-389. – Предм. указ.: с. 390-395. – 4700 экз. (в пер.).

10 Новиков, Ф. А. Дискретная математика для программистов [Текст]: учеб. для вузов / Ф. А. Новиков. - 2-е изд. - СПб.: Питер, 2004. - 364 с.: ил. - Библиогр.: с. 349-350. - Предм. указ.: с. 351-363. – ISBN 5-94723-741-5.

11 Гордеев, А. В. Системное программное обеспечение [Текст]: учеб. для вузов / А. В. Гордеев, А. Ю. Молчанов; под. общ. ред. А. В. Гордеева. – СПб.: Питер, 2001. - 734 с.: ил.; 24 см. - Библиогр.: с. 719-724. – 5000 экз. – ISBN 5-272-00341-1 (в пер.).

12 Хоар, Ч. Взаимодействующие последовательные процессы [Текст] / Хоар Чарльз; перевод с англ. А. А. Бульонковой; под. общ. ред. А. П. Ершова. - М.: Мир, 1989. - 264 с.: ил.; 22 см. - Библиогр.: с. 254. - Предм. указ.: с. 259-262. – Перевод изд.: Communicating sequential processes / C.A.R. Hoare. - Englewood Cliffs etc. – 12300 экз. – ISBN 5-03-001043-2.

13 Ключко, В. И. Теория вычислительных процессов и структур [Текст]: учеб. пособие для студентов спец. 220400 и аспирантов кафедры ВТ и АСУ / В. И. Ключко; М-во общ. и проф. образования РФ, Кубан. гос. технол. ун-т. - Краснодар: Изд-во КубГТУ, 1999. - 118 с.: ил.; 20 см. - Библиогр.: с. 117. – 75 экз. – ISBN 5-230-21918-1.

14 Котов, В. Е. Сети Петри [Текст] / В. Е. Котов. – М.: Наука, 1984. - 158 с.: ил.; 22 см. - Библиогр.: с. 150-152. – 5800 экз. (в пер.).

15 Питерсон, Дж. Теория сетей Петри и моделирование систем [Текст] / Питерсон Джеймс; перевод с англ. М. В. Горбатовой и др.; под общ. ред. д. т. н. В. А. Горбатова. - М.: Мир, 1984. - 264 с.: ил.; 22 см. - Библиогр.: с. 234-261. - Предм. указ.: с. 262-263. - Перевод изд.: Petri net theory and the modeling of sistems / James L. Peterson . - Prentice-Hall, 1981. – 8400 экз.