СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Проблема создания высокоэффективных устройств логического управления является актуальной благодаря качественному изменению характера объектов логического управления, невозможности построения управляющих устройств на основе традиционных решений, существенному расширению класса решаемых задач, появлению новой элементной базы (программируемые логические интегральные схемы (ПЛИС)). Большинство задач проектирования подобных устройств относятся к комбинаторно-топологическим и характеризуются высокой трудоемкостью при отсутствии эффективных методов и алгоритмов решения.

Одной из эффективных ПЛИС-реализаций устройств логического управления функционально сложными объектами являются микроконтроллерные сети, известные также как микропрограммные мультимикроконтроллеры (МПММК). МПММК представляют собой параллельные структуры, формируемые из множества однотипных СБИС – микропрограммных микроконтроллеров (МПМК). Реализация алгоритмов управления в МПММК не допускает непосредственное использование методов, разработанных в рамках теории дискретных управляющих устройств и микропрограммных автоматов. Таким образом, актуальной представляется разработка оригинальных моделей функционально-структурной организации МПММК, а также общей методологии синтеза устройств на их основе.

В работе освещена часть этапа морфологического синтеза МПММК, включающего комплекс задач, направленных на оптимальное представление алгоритмов логического управления в виде сети компонентных алгоритмов, распределенных между отдельными модулями микроконтроллера. Задача выбора разбиения имеет ярко выраженный комбинаторно-топологический характер. Алгоритм ее решения, представленный в [1], учитывает ряд ограничений, состав и содержание которых зависят от архитектуры СБИС МПМК, технологических ограничений, структурной организации МПММК, дисциплины взаимодействия микроконтроллеров и др. К их числу следует отнести: минимальное

число подалгоритмов, ограниченная сложность подалгоритмов, минимальная сложность сети межблочных взаимодействий и т.д. Постановка задачи представляет собой наиболее общий случай и в зависимости от функционально-топологической организации МПММК может меняться.

Суть параллельно-последовательного алгоритма формирования субоптимальных разбиений сводится к следующему. Исходная ПарГСА G^0, удовлетворяющая набору ограничений, приводится к ациклическому виду с использованием $\overline{\omega}$-преобразования. Затем ПарГСА ставится в соответствие система выражений Ξ, которая используется для формирования сечений (сечением является группа вершин ПарГСА, элементы которой не находятся в отношении связи). Первоначально формируется базовое сечение с использованием *R*-алгоритма, после чего начинается перебор смежных сечений “вверх” (получение *u*-сечений) и “вниз” (получение *d*-сечений). Полученное разбиение алгоритма на сечения используется при формировании блоков разбиения, включающих в себя подалгоритмы исходной ПарГСА, которые могут быть выполнены одним МПМК.

1. ПОСТАНОВКА ЗАДАЧИ ПОИСКА МАКСИМАЛЬНОГО СЕЧЕНИЯ АЛГОРИТМА

Одним из этапов параллельно-последовательного метода формирования субоптимальных разбиений является поиск максимального (базового) сечения ПарГСА, заданной в виде системы выражений Ξ. Исходная система Ξ считается корректной (соответствующей ограничениям, описанным в [1]), предикаты условий, не влияющие на процесс поиска базового сечения, опущены. В результате преобразования исходной системы Ξ по правилам *R*-алгоритма получается система Ξ^*, содержащая два выражения вида

$$\begin{array}{l} a_0 \to R_\gamma \\ R_\gamma \to a_k \end{array}, \qquad (1)$$

причем R_γ является искомым базовым сечением. Найденное максимальное сечение, представленное в конструктивной (скобочной) форме, предполагается использовать в дальнейшем при организации параллельно-последовательного перебора смежных сечений.

Преобразование системы с использованием *R*-алгоритма заключается в применении одного из трех правил (*u*-поглощение, *d*-поглощение, ψ-перегруппировка), в результате чего система видоизменяется. Применение правил *u*- и *d*-поглощения ведет к уменьшению числа выражений на единицу, ψ-перегруппировка не изменяет количества выражений системы и предназначается для изменения структуры одного из выражений с целью проведения *d*-поглощения с его участием на следующей итерации алгоритма. Если исходная система выражений корректна, то при помощи правил *R*-алгоритма она приводится к виду (1), после чего выделяется базовое сечение. В противном случае в системе остается некоторое количество выражений $N>2$, причем к ней невозможно применить ни одно правило *R*-алгоритма и она является т.н. нередуцируемой ω-формой.

Проведение преобразований системы вручную затруднительно, поэтому представляет интерес автоматизация этого процесса с использованием ЭВМ.

2. СТРУКТУРНАЯ СХЕМА ЭТАПА

Исходными данными для этапа поиска базового сечения являются система *R*-выражений и матрица отношений между вершинами (рис. 1). Результат выполнения этапа – базовое сечение алгоритма и протокол преобразований.

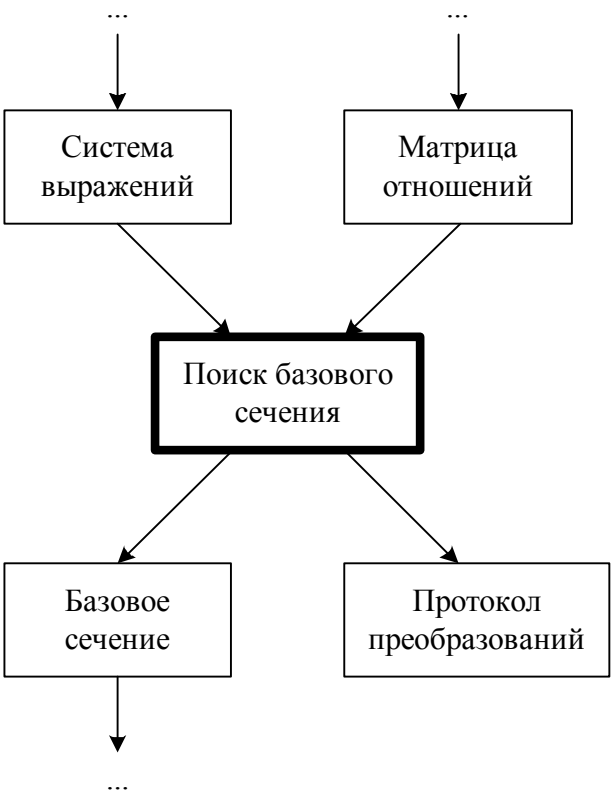


Рис.1 Структурная схема этапа

3. РАЗРАБОТКА СТРУКТУР ДАННЫХ

Алгоритм нахождения базового сечения ориентирован на преобразование исходной системы выражений вида

$$R_1^i \to R_2^i, \quad i = \overline{1, n}, \tag{2}$$

согласно правилам *R*-алгоритма. Система задается в виде динамического массива выражений:

```
TKsiSystem = record
  Items: array of TKsiSystemItem;
  VertexCount: Integer;
  RM: TRelationMatrix;
end;
```

где `Items` – массив выражений системы;

`VertexCount` – количество вершин алгоритма (нумерация вершин с 0 до `VertexCount-1`, начальной считается вершина a_0, конечной – $a_{\texttt{VertexCount-1}}$);

`RM` – матрица отношений (необходима для устранения неоднозначностей при выполнении ψ-перегруппировки).

Элемент системы выражений представляет собой совокупность двух *R*-выражений:

```
TKsiSystemItem = record
  R1, R2: TRExpr;
end;
```

R-выражения представляются в виде деревьев. Узлами дерева являются вершины, хранящие информацию о том, в каком отношении находятся их потомки ("|" – отношение альтернативы, "•" – отношение параллельности). Листьями дерева являются подмножества множества вершин алгоритма $\{a_0, a_1, ..., a_{VertexCount-1}\}$. Например, *R*-выражение $a_7 \mid a_8 \mid (a_9 \bullet a_{10})$ будет представлено в виде дерева следующим образом:

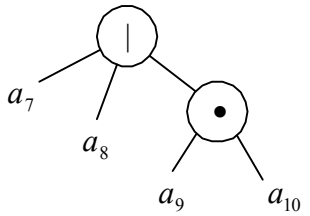


Рис.2 Пример дерева, соответствующего R-выражению

В программе деревья (*R*-выражения) хранятся в виде массива записей следующего вида:

```
TRExpr = array of TExprItem;

TExprItem = record
  ExprType: TExprType;
  DownLinks: TBits;
  UpLink: Integer;
end;
```

где `ExprType` – тип узла (`etLsts` – набор листьев, `etParall` – отношение параллельности, `etAltern` – отношение альтернативы);

`DownLinks` – ссылки на потомков узла;

`UpLink` – ссылка на предка узла.

Взаимосвязь между узлами дерева осуществляется путем хранения ссылок на предка (UpLink'а) и потомков (DownLink'ов). Ссылка представляет собой номер позиции элемента в массиве. Связи между элементами дерева организованы таким образом, что возможно движение по дереву в двух направлениях: от корня к листьям ("сверху-вниз") и от листьев к корню ("снизу-вверх"). Листья, являющиеся потомками одного узла, занимают один элемент массива и хранятся в виде множества. Стандартные множества, предоставляемые Делфи (`set`), ограничены 255 элементами, поэтому для реализации множества используется класс `TBits`, входящий в состав VCL. Если длина массива `TRExpr` равна L, то элементы дерева хранятся в позициях с 0 по L–1, причем корень дерева должен находиться в позиции L–1. Поле `UpLink` для корня дерева должно быть отрицательным.

Примеры представления R-выражений в виде массива `TRExpr`:

№	Тип элемента (`ExprType`)	Множество вершин / ссылок (`DownLinks`)	Ссылка на предка (`UpLink`)
$a_7 \mid a_8 \mid (a_9 \bullet a_{10})$			
0	L	$\{a_9, a_{10}\}$	1
1	•	$\{r_0\}$	3
2	L	$\{a_7, a_8\}$	3
3	|	$\{r_1, r_2\}$	–1
$a_2 \bullet a_5 \bullet a_6 \bullet (a_7 \mid a_8 \mid (a_9 \bullet a_{10}))$			
0	L	$\{a_9, a_{10}\}$	1
1	•	$\{r_0\}$	3
2	L	$\{a_7, a_8\}$	3
3	|	$\{r_1, r_2\}$	5
4	L	$\{a_2, a_5, a_6\}$	5
5	•	$\{r_3, r_4\}$	–1

В данном случае через r_i обозначены ссылки на потомков, через a_i – номера вершин алгоритма в наборах листьев.

4. ВВОД СИСТЕМЫ ВЫРАЖЕНИЙ

Представление *R*-выражений в виде деревьев в программе довольно громоздко с т.з. человека, поэтому не представляется возможным задание выражений системы, например, в виде констант. Для этого в программу включен модуль `Parser` и средства обработки текстовых файлов определенного формата, при помощи которых реализуется ввод данных и осуществляет преобразование текстового представления системы в набор взаимосвязанных деревьев. Пример входного файла:

```
[Size]
3

[VertexCount]
```

```
13

[System]
a0 -> (a1|a2)*(a3|a4)*(a5|a6)
a1*a3*a5 -> (a7|a8)*a9*a10*a11
(a7|a8)*a9*a10*a11 -> a12

[RelationMatrix]
a1    a3    *
a1    a5    *
a3    a5    *
a2    a4    *
a2    a6    *
a4    a6    *
```

Секция	Назначение
Size	Размер системы
VertexCount	Количество вершин алгоритма
System	Система выражений
RelationMatrix	Матрица отношений

Ввод данных из файла является промежуточным этапом, поэтому он реализован максимально упрощенно: порядок следования секций фиксирован, файл считается синтаксически корректным (дополнительных проверок корректности не производится). Допускается использование разделителей в виде пробелов и символов табуляции, а также вставка пустых строк между секциями.

В дальнейшем система выражений будет получаться в процессе преобразования граф-схемы алгоритма на одном из нереализованных в данный момент этапов, и необходимость в процедурах и функциях модуля `Parser` по видимому отпадет.

5. РАЗРАБОТКА АЛГОРИТМОВ ЭЛЕМЕНТАРНЫХ ОПЕРАЦИЙ НАД *R*-ВЫРАЖЕНИЯМИ

Преобразование системы выражений сводится к применению правил *R*-алгоритма. Правила *u*-, *d*-поглощения и *ψ*-

перегруппировки допускают разбиение на более простые операции над *R*-выражениями. К их числу относятся:

- проверка отношения нестрогого включения $R_i[\subseteq]R_j$;
- сравнение субсечений (поддеревьев);
- проверка принадлежности вершины a_i сечению R_j;
- вычисление ω-мощности сечения;
- раскрытие скобок в *R*-выражениях;
- проверка структуры перегруппировываемого дерева (*RT*-дерева);
- проверка структуры дерева, под которое происходит перегруппировка (*ST*-дерева);
- получение перегруппированного подвыражения;
- подстановка деревьев;
- удаление “лишних” элементов из дерева.

Каждая из перечисленных операций заслуживает детального рассмотрения.

5.1. Проверка принадлежности вершины a_i сечению R_j

Для проверки принадлежности вершины сечению необходимо найти среди элементов дерева такой набор листьев, в состав которого входит искомая вершина. Если такой набор найден, то вершина входит в состав сечения, в противном случае – не входит. Проверка осуществляется функцией `AiInRExpr()` модуля `RExprs`.

5.2. Вычисление ω-мощности сечения

Для вычисления ω-мощности *R*-выражения необходимо организовать движение по дереву от корня к листьям (сверху вниз), при этом вычисление сводится к проверке типа узла и выполнению определенных действий:

Тип узла дерева	Алгоритм вычисления ω-мощности
|	Максимальная ω-мощность подвыражения
•	Сумма ω-мощностей подвыражений
L	Количество листьев в наборе

Примеры R-выражений и их ω-мощности:

Выражение	ω-мощность
$a_1 \mid a_2$	1
$a_1 \bullet a_2$	2
$a_1 \mid (a_2 \bullet a_3) \mid a_4$	2
$a_1 \bullet (a_2 \mid a_3) \bullet a_4$	3

Движение по дереву происходит рекурсивно (вложенная функция `GetWPowerRecurse()`). Вычисление ω-мощности осуществляется функцией `GetWPower()` модуля `RExprs`.

5.3. Сравнение субсечений (поддеревьев)

Деревья считаются эквивалентными, если они имеют одинаковую структуру, т.е. каждому узлу одного дерева взаимно однозначно соответствует узел другого дерева, причем узлы совпадают по типу и по структуре поддеревьев, корнями которых они являются. Сравнение поддеревьев осуществляется по следующему алгоритму:

1. Если корневые элементы не равны по типу, то деревья не эквивалентны. Перейти к п.4.
2. Если корневыми элементами являются наборы листьев, то сравнить их и на основании сравнения сделать вывод об эквивалентности деревьев. Перейти к п.4.

3. Среди множества потомков деревьев найти эквивалентные поддеревья и исключить их из числа потомков. Если останутся несоответствующие друг другу потомки, то деревья не эквивалентны.
4. Конец алгоритма.

Алгоритм сравнения деревьев реализует рекуррентная функция `EqualTreesRecurse()`.

5.4. Проверка отношения нестрогого включения

В процессе поиска деревьев для *u*- или *d*-поглощения возникает необходимость в выяснении того, является ли одно дерево полностью или частично поддеревом другого, и определении того, какая часть дерева полностью эквивалентна, а какая полностью неэквивалентна. Дерево *A* неполностью эквивалентно дереву *B* в том случае, если дерево *A* является поддеревом дерева *B*, однако у элемента дерева *B*, соответствующего корню дерева *A*, имеются потомки, не входящие в состав дерева *A*. При реализации подстановок деревьев неэквивалентные потомки остаются у корня, поэтому в программе они получили наименование `StayedDownLinks`. Рассмотрим несколько примеров деревьев, находящихся в отношении неполной эквивалентности (рис. 3, 4).

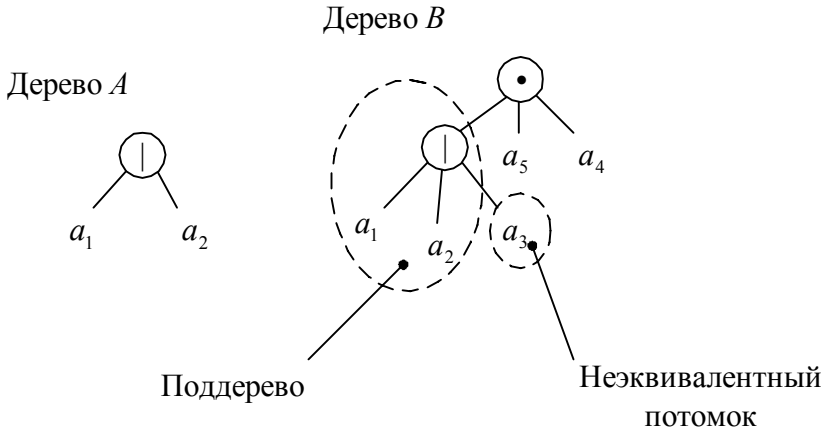


Рис.3. Неполная эквивалентность деревьев

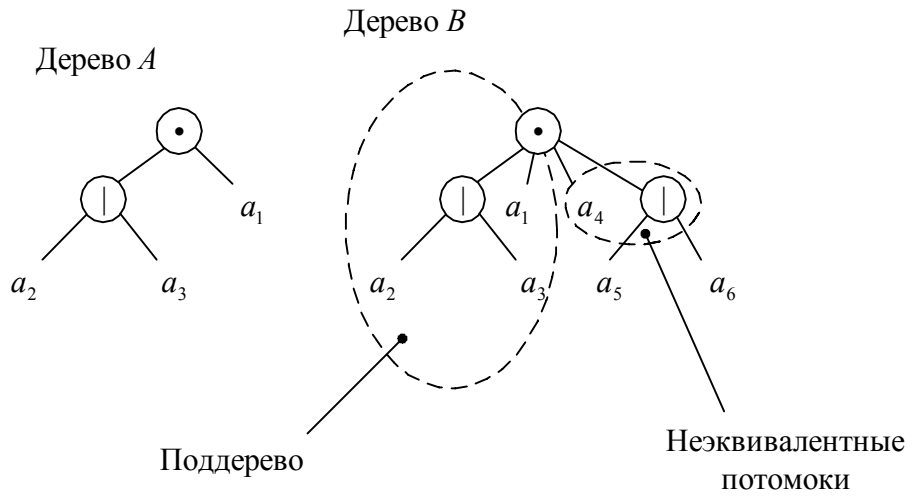


Рис.4. Неполная эквивалентность деревьев

Проверка деревьев на предмет неполной эквивалентности производится рекуррентной функцией `EqualTrees()`.

Алгоритм проверки неполной эквивалентности в целом похож на алгоритм сравнения деревьев. Отличие состоит в том, что если после поиска и удаления эквивалентных потомков в *R2* (дерево *B*) останутся элементы, то это говорит о том, что деревья не полностью эквивалентны, а оставшиеся элементы – ссылки на неэквивалентных потомков (`StayedDownLinks`).

Проверка отношения нестрогого включения *R*-выражений заключается в нахождении такого узла в дереве *B*, который является эквивалентом корня дерева *A*. Причем помимо нахождения номера узла для дальнейших операций необходима информация о полной/неполной эквивалентности и, в случае неполной эквивалентности, о неэквивалентных потомках.

Проверку можно организовать следующим образом: проверить все *N* узлов дерева *B* на предмет нахождения того, который эквивалентен корню дерева *A*. Однако в этом случае необходимо провести *N* сравнений деревьев (наихудший вариант, который может возникнуть, например, при сравнении неэквивалентных деревьев), что не очень эффективно для больших деревьев ввиду рекуррентного характера процедур сравнения. Поэтому необходимо каким-либо образом уменьшить число рассматриваемых узлов дерева *B* либо изменить процедуру сравнения деревьев. В программе реализован первый вариант.

Сопоставим каждой вершине дерева B числовую оценку. Вершину будем считать кандидатом в эквивалент корня дерева, если оценка вершины и оценка корня дерева *A* совпадают. Сформулируем алгоритм формирования оценок:

1. Первоначально оценки всех узлов дерева *B* принимаются равными нулю.
2. Вычисляется оценка корня дерева *A* как количество всех возможных путей от листьев дерева к корню.
3. Для всех узлов дерева *A*, которые являются листьями, проделывается следующая операция: восстанавливается маршрут от листа к корню в виде последовательности типов пройденных узлов (“•”, “|”, “•”, “•”) в направлении снизу-вверх, после чего в дереве *B* находятся все аналогичные маршруты, и оценки вершин, в которых происходит окончание маршрута, увеличиваются на единицу.
4. Из дерева *B* выбираются вершины, оценки которых совпадают с оценкой корня дерева *A*, и для них проводится рекурсивное сравнение деревьев.

Рассмотрим алгоритм формирования оценок на примере (рис.5).

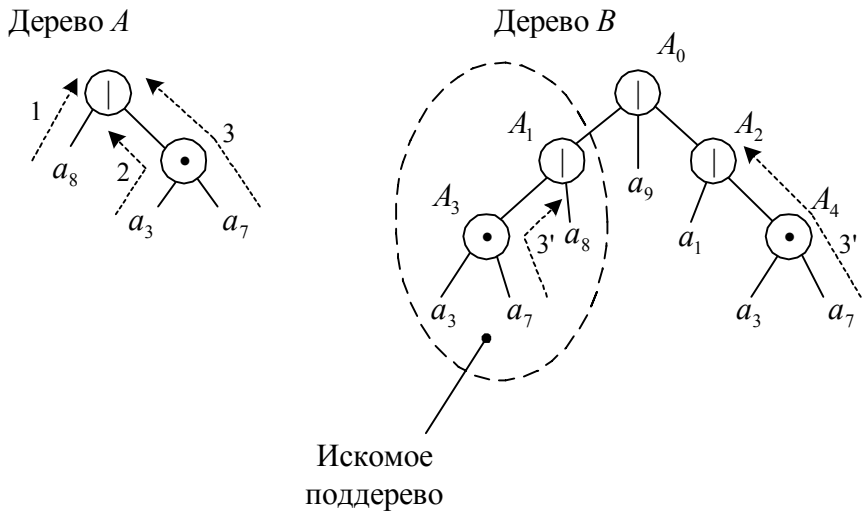


Рис.5. Формирование оценок

Несложно заметить, что корню дерева *A* будет соответствовать оценка 3, т.к. дерево содержит 3 листа и, следовательно, 3 пути сходятся в корне. Сформируем оценки для узлов дерева *B*.

Через A_i обозначим номера вершин дерева B. Начнем рассмотрение маршрутов в дереве A:

Начальная вершина	Маршрут в дереве A	Номера вершин маршрута в дереве B	Окончание маршрута	Оценки
				m_0=0 m_1=0 m_2=0 m_3=0 m_4=0
a_8	“|”	a_8– A_1	A_1	m_0=0 m_1=1 m_2=0 m_3=0 m_4=0
a_3	“•”, “|”	a_3–A_3–A_1	A_1	m_0=0 m_1=2 m_2=0 m_3=0 m_4=0
		a_3–A_4–A_2	A_2	m_0=0 m_1=2 m_2=1 m_3=0 m_4=0
a_7	“•”, “|”	a_7–A_3–A_1	A_1	m_0=0 m_1=3 m_2=1 m_3=0 m_4=0
		a_7–A_4–A_2	A_2	m_0=0 m_1=3 m_2=2 m_3=0 m_4=0

Кандидат для более детальной проверки – вершина A_1.

Для вершин-кандидатов необходимо проведение дополнительной детальной рекурсивной проверки с целью выяснения структуры дерева. Без рекурсивной проверки обойтись невозможно, т.к. возможны варианты деревьев (рис.6), в которых оценка вершины будет соответствовать оценке корня, однако деревья не будут эквивалентны.

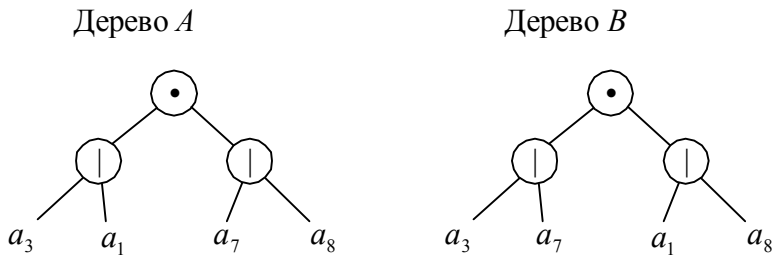


Рис.6 Совпадение оценок в неэквивалентных деревьях

В приведенном примере корневые элементы имеют оценки 4, однако деревья не эквивалентны.

Механизм оценок является своего рода аналогом хэш-функций с тем отличием, что в качестве хэш-функции выступает алгоритм формирования оценок. Введение системы оценок позволяет сузить круг вершин-кандидатов в эквивалент корня, что должно положительно сказаться на скоростных характеристиках программы.

Проверка отношения нестрогого включения выполняется процедурой `IncludedRExprs()`.

5.5. Раскрытие скобок в R-выражениях

Иногда после подстановок *R*-выражений возникает необходимость раскрывать скобки. Например, если в результате выполнения подстановки получается выражение $a_1 \bullet (a_2 \bullet a_3 \bullet (a_4 \mid a_5))$, то оно должно быть преобразовано к виду $a_1 \bullet a_2 \bullet a_3 \bullet (a_4 \mid a_5)$.

С точки зрения преобразования деревьев раскрытие скобок сводится к следующему: необходимо найти такую пару узлов предок-потомок, которые имеют одинаковый тип. Результатом преобразования является поглощение потомка предком, при этом все DownLink’и потомка переходят к предку.

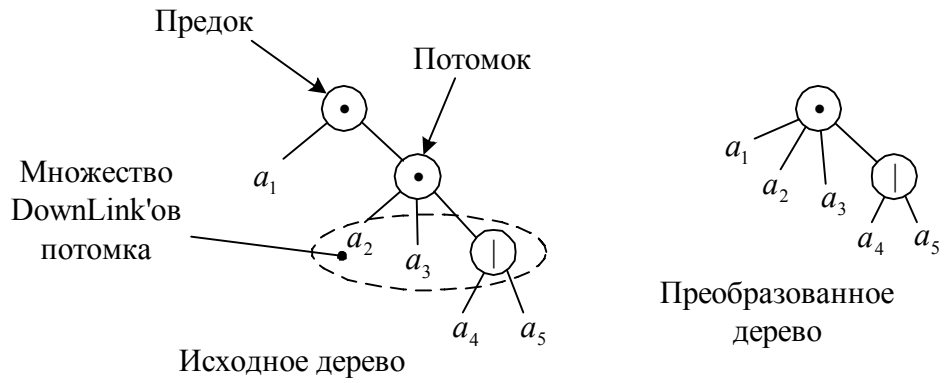


Рис.7 Преобразование деревьев при раскрытии скобок

Алгоритм преобразования:

1. Найти пару предок-потомок. Если пара не найдена, то п.4
2. DownLink’и потомка переходят к предку (перенастройка связей)
3. Удаление потомка из дерева. Переход к п.1
4. Конец алгоритма

Выполнение преобразования осуществляется процедурой `TransformRExpr()`.

В результате удаления элемента из дерева образуется пустое место (элемент массива, который не связан с другими элементами). Для удаления таких неиспользуемых элементов массива предназначена процедура `CheckRExprForEmptys()`.

5.6. Удаление элементов из дерева

При модификации деревьев (подстановки деревьев, раскрытие скобок) возникает необходимость в реорганизации структуры дерева и удалении части его узлов. Удаление элемента заключается в освобождении занимаемой им динамической памяти, однако позиция, занимаемая им в массиве остается, хотя фактически не используется. Удаленные элементы однозначно идентифицируются по отсутствию связей с другими элементами дерева. Наличие таких элементов в дереве (точнее, в массиве, который хранит дерево) может негативно сказаться на работе процедур преобразования деревьев: может вызвать ложные результаты проце-

дур проверки структуры деревьев, возбуждение исключительных ситуаций, что крайне нежелательно. Поэтому возникает необходимость в удалении неиспользуемых элементов из массива.

После нахождения неиспользуемых позиций массива происходит сдвиг используемых элементов в массиве в сторону младших элементов и корректировка связей, затем корректируется длина массива, хранящего дерево.

Описанное преобразование выполняется процедурой `CheckRExprForEmptys()`.

5.7. Подстановка деревьев

Необходимость в подстановке деревьев возникает при выполнении операций *u*-, *d*-поглощения и ψ-перегруппировки. В результате проверки пригодности структуры деревьев для проведения какой-либо операции выясняется, какой узел является эквивалентом корня, находятся ли поддерево и искомое дерево в отношении полной эквивалентности, определяется группа узлов, которые останутся у эквивалента корня после подстановки. Основная сложность данной операции заключается в том, что практически в каждом из шести частных случаев, рассматриваемых ниже, изменение структуры дерева происходит с некоторыми индивидуальными особенностями. Рассмотрим эти частные случаи:

№	Удаляемый элемент	Вставляемый элемент	Соответствие	Индекс UpLink'а при настройке перекрестных связей	Рекурсивное удаление поддерева	Копирование дерева	Удаление части листьев	Добавление части листьев
1	лист	лист	полное	–	–	–	+	+
2	лист	дерево	полное	UpLink листьев	–	+	+	–
3	дерево	дерево	полное	UpLink корня удаляемого дерева	+	+	–	–
4	лист	лист	частичное	–	–	–	+	+
5	лист	дерево	частичное	UpLink листьев	–	+	+	–
6	дерево	дерево	частичное	Корень удаляемого дерева	+	+	+	+

Примечание. Сочетание дерево-лист не встречается при подстановках, т.к. такая подстановка ведет к уменьшению ω-мощности результирующего R-выражения, что противоречит условиям преобразования.

Примеры систем выражений для иллюстрации всех частных случаев (подстановка возникает при выполнении операции *и*-поглощения):

1. Лист-Лист-Полное соответствие (сокращенно л-л-п): $\begin{cases} a_1 \to a_2 \\ a_2 \to a_3 \end{cases} \Rightarrow \quad a_1 \to a_3$

2. л-д-п: $\begin{cases} a_1 \to a_2 \\ a_2 \to a_3 \bullet a_4 \end{cases} \Rightarrow \quad a_1 \to a_3 \bullet a_4$

3. д-д-п: $\begin{cases} a_1 \to a_2 \bullet a_3 \\ a_2 \bullet a_3 \to a_4 \mid a_5 \end{cases} \Rightarrow \quad a_1 \to a_4 \mid a_5$

4. л-л-ч: $\begin{cases} a_1 \to a_2 \mid a_3 \\ a_2 \to a_4 \end{cases} \Rightarrow \quad a_1 \to a_4 \mid a_3$

5. л-д-ч: $\begin{cases} a_1 \to a_2 \mid a_3 \\ a_2 \to a_4 \bullet a_5 \end{cases} \Rightarrow \quad a_1 \to (a_4 \bullet a_5) \mid a_3$

6. д-д-ч: $\begin{cases} a_0 \to a_1 \mid a_2 \mid (a_3 \bullet a_4) \\ a_2 \mid (a_3 \bullet a_4) \to a_5 \bullet a_6 \bullet a_7 \end{cases} \Rightarrow \quad a_0 \to a_1 \mid (a_5 \bullet a_6 \bullet a_7)$

Из таблицы можно сформулировать несколько правил, касающихся алгоритма модификации дерева:

- Настройка значения UpLink'а не требуется при подстановках л-л-*, в остальных случаях индивидуальна;
- Рекурсивное удаление поддерева происходит только если удаляемый элемент – дерево;
- Копирование дерева происходит только если вставляемый элемент – дерево;
- Удаление части листьев происходит всегда, кроме подстановки д-д-п;
- Добавление части листьев происходит при подстановках л-л-* и д-д-ч.

После подстановки необходимо проверить полученное дерево на наличие скобок и раскрыть их. Подстановки деревьев осуществляются процедурой `DoSubstRExprs()`.

5.8. Проверка структуры дерева, под которое происходит перегруппировка

Будем называть дерево, под которое происходит перегруппировка, *ST*-деревом (Sub Tree). Дерево является кандидатом в *ST*-дерево в том случае, когда оно имеет структуру типа

$$(a_{11} \bullet a_{12} \bullet \ldots \bullet a_{1n}) \mid (a_{21} \bullet a_{22} \bullet \ldots \bullet a_{2n}) \mid \ldots \mid (a_{m1} \bullet a_{m2} \bullet \ldots \bullet a_{mn}) \qquad (3)$$

или то же самое в виде дерева:

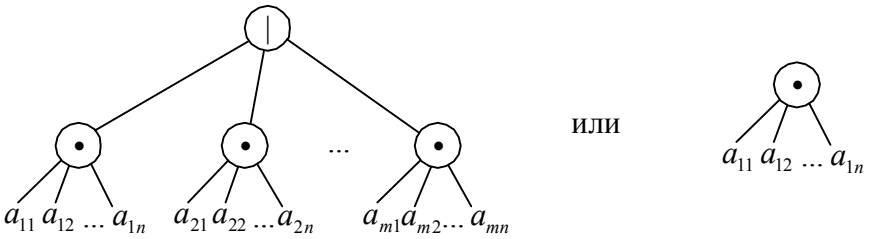


Рис.8 Структура ST-дерева

Для проверки структуры дерева необходимо выполнение следующих условий:

1. Корень дерева должен иметь тип "|" (допускается наличие корня типа "•" в том случае, когда в состав дерева входит всего одна группа, т.е. m=1)
2. Корень содержит только потомков типа "•"
3. Каждый потомок корня содержит только потомков типа листья, причем количество листьев в каждом наборе фиксировано и равно n.

Кроме проверки структуры дерева производится подсчет количества групп (скобок) G и листьев в скобке L. В нашем случае $G=m$, $L=n$.

Для того чтобы различать обозначения количества элементов *ST*- и *RT*-деревьев, будем снабжать символы G и L подстрочными индексами: G_{RT}, L_{ST}.

Рассмотрим примеры *ST*-деревьев:

№	Выражение	Результат проверки	G_{ST}	L_{ST}
1	a_0	–	–	–
2	$a_1 \bullet a_2$	+	1	2
3	$a_1 \bullet a_2 \bullet a_3$	+	1	3
4	$(a_1 \bullet a_2) \mid (a_3 \bullet a_4)$	+	2	2
5	$a_1 \mid a_2$	–	–	–
6	$(a_1 \mid a_2) \mid (a_3 \bullet a_4)$	–	–	–
7	$(a_1 \bullet a_2) \mid (a_3 \bullet a_4 \bullet a_5)$	–	–	–
8	$(a_1 \bullet a_2 \bullet a_3) \mid (a_4 \bullet a_5 \bullet a_6)$	+	2	3
9	$(a_1 \bullet a_2) \mid (a_3 \bullet a_4) \mid (a_5 \bullet a_6)$	+	3	2

Выполнение проверки структуры *ST*-деревьев осуществляется функцией `PsiRegr_CheckForSubTree()`.

5.9. Проверка структуры перегруппировываемого дерева

Будем называть дерево, в котором планируется осуществлять перегруппировку, *RT*-деревом (Regrouped Tree). Дерево является кандидатом в *RT*-дерево, если оно имеет структуру вида

$$\tilde{R} \circ ((a_{11} \mid a_{21} \mid ... \mid a_{m1}) \bullet (a_{12} \mid a_{22} \mid ... \mid a_{m2}) \bullet ... \bullet (a_{1n} \mid a_{2n} \mid ... \mid a_{mn})) \qquad (4)$$

или то же самое в виде дерева:

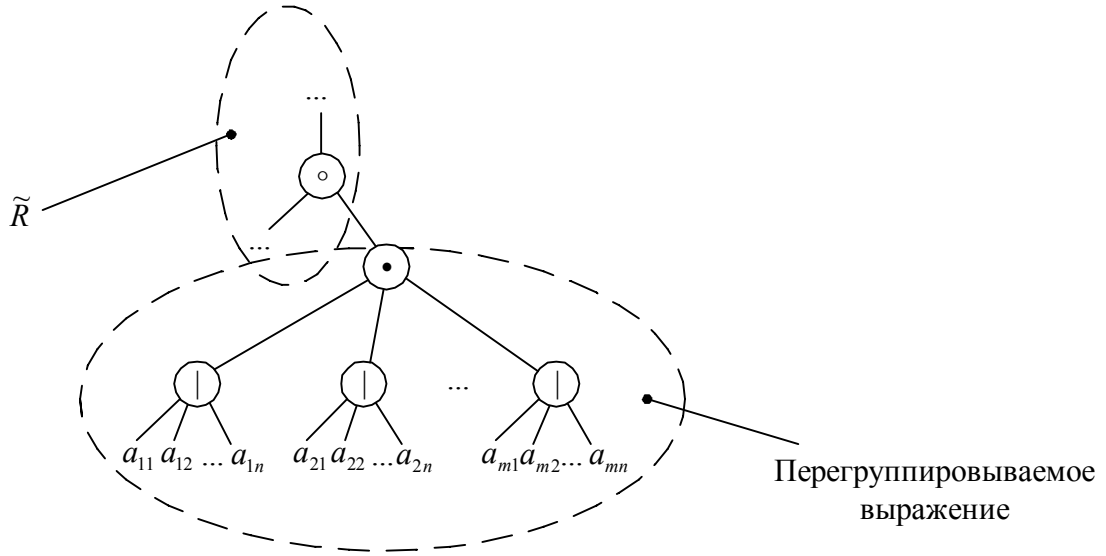


Рис.9 Структура RT-дерева

т.е. содержит поддерево, удовлетворяющее следующим требованиям:

1. Корень поддерева – имеет тип “•”
2. Корень поддерева имеет G_{RT} потомков типа “|”
3. Каждый из потомков корня типа “|” имеет только потомка типа листья, причем количество листьев одинаково и равно L_{RT}.

Замечание. В отличие от *ST*-дерева, корень поддерева *RT*-дерева может содержать потомков, которые не участвуют в перегруппировке (неполная эквивалентность деревьев).

Для выполнения операции ψ-перегруппировки между деревьями должны выполняться следующие соотношения:

$$\begin{matrix} G_{ST} \le L_{RT} \\ L_{ST} = G_{RT} \end{matrix}, \tag{5}$$

в противном случае считается, что деревья не подходят для перегруппировки.

Процесс проверки *RT*-дерева выполняется функцией `PsiRegr_CheckForRegrTree()`.

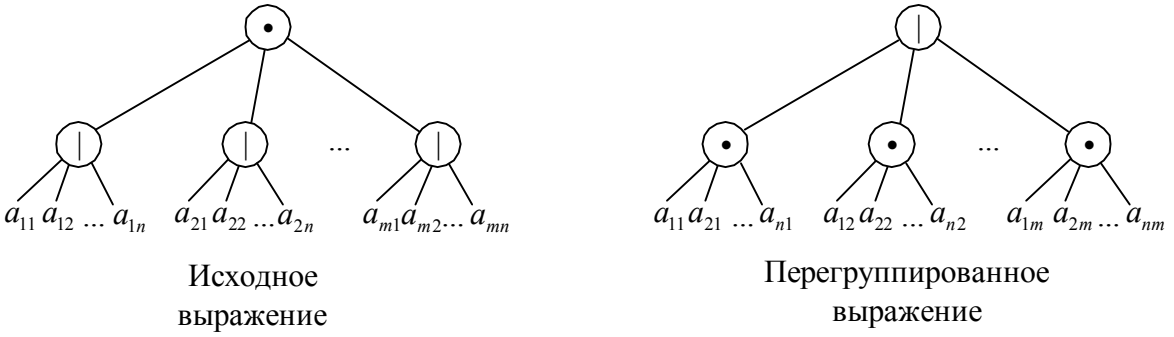
5.10. Получение перегруппированного выражения

Перегруппированное выражение получается из выражения вида

$$(a_{11} \mid a_{21} \mid ... \mid a_{m1}) \bullet (a_{12} \mid a_{22} \mid ... \mid a_{m2}) \bullet ... \bullet (a_{1n} \mid a_{2n} \mid ... \mid a_{mn}) \tag{6}$$

путем изменения его структуры:

$$(a_{11} \bullet a_{12} \bullet ... \bullet a_{1n}) \mid (a_{21} \bullet a_{22} \bullet ... \bullet a_{2n}) \mid ... \mid (a_{m1} \bullet a_{m2} \bullet ... \bullet a_{mn}) \tag{7}$$



Исходное выражение

Перегруппированное выражение

Рис.10 Результат перегруппировки

Однако изменение структуры дерева само по себе неоднозначно. Например, выражение $(a_1 \mid a_2) \bullet (a_3 \mid a_4) \bullet (a_5 \mid a_6)$ может быть перегруппировано как $(a_1 \bullet a_3 \bullet a_5) \mid (a_2 \bullet a_4 \bullet a_6)$ или как $(a_1 \bullet a_3 \bullet a_6) \mid (a_2 \bullet a_4 \bullet a_5)$. С точки зрения рассматриваемой системы выражений только один вариант перегруппировки является правильным. Для устранения неоднозначности используется матрица

отношений, по которой однозначно идентифицируется набор вершин, входящих в одну скобку (данные вершины находятся в отношении параллельности “•”).

Алгоритм получения перегруппированного дерева:

1. Формирование наборов листьев (для формирования набора из каждой скобки исходного выражения берется 1 вершина, причем все вершины должны находиться между собой в отношении параллельности).
2. Надстройка структуры дерева: набора листьев объединяются отношением параллельности, а отношения параллельности в свою очередь – отношением альтернативы.

Процесс формирования перегруппированного выражения осуществляется функцией `GetRegrRExpr()`.

6. РЕАЛИЗАЦИЯ ОПЕРАЦИЙ *R*-АЛГОРИТМА

6.1. U-поглощение

В системе выражений Ξ находятся выражения i и j, которые удовлетворяют следующим условиям:

1. Номера выражений различны
2. Конечная вершина не входит в правую часть j-го выражения
3. ω-мощность правой части j-го выражения не меньше ω-мощности левой части j-го
4. Левая часть j-го выражения является конструктивным подмножеством правой части i-го

Замечание. С целью повышения эффективности выполнения u-поглощения проверка условий осуществляется именно в таком порядке, как описано выше.

Если все условия выполнены, то производится подстановка правой части j-го выражения в левую часть i-го и удаление j-го выражения из системы.

Примеры u-поглощений:

1. $\begin{cases} a_0 \to a_2 \\ a_2 \to a_1 \mid a_3 \end{cases} \Rightarrow \quad a_0 \to a_1 \mid a_3$

2. $\begin{cases} a_0 \to a_1 \mid a_3 \\ a_1 \to a_4 \mid a_5 \end{cases} \Rightarrow \quad a_0 \to a_1 \mid a_4 \mid a_5$

3. $\begin{cases} a_0 \to a_1 \mid a_4 \mid a_6 \mid a_{11} \\ a_6 \mid a_{11} \to a_{14} \bullet (a_{13} \mid a_{69}) \end{cases} \Rightarrow \quad a_0 \to a_1 \mid a_4 \mid (a_{14} \bullet (a_{13} \mid a_{69}))$

Описанные действия осуществляются функцией `DoUSubst()`.

6.2. D-поглощение

В системе выражений Ξ находятся выражения *i* и *j*, которые удовлетворяют следующим условиям:

1. Номера выражений различны
2. Начальная вершина не входит в левую часть *i*-го выражения
3. ω-мощность левой части *i*-го выражения не меньше ω-мощности правой части *i*-го
4. Правая часть *i*-го выражения является конструктивным подмножеством левой части *j*-го

Если все условия выполнены, то производится подстановка правой части *i*-го выражения в левую часть *j*-го и удаление *i*-го выражения из системы.

Примеры *u*-поглощений:

1. $\begin{cases} a_{45} \bullet a_{47} \to a_{54} \\ a_{54} \mid (a_{43} \bullet a_{74}) \to a_{60} \bullet a_{63} \end{cases} \Rightarrow \quad (a_{45} \bullet a_{47}) \mid (a_{43} \bullet a_{74}) \to a_{60} \bullet a_{63}$

2. $\begin{cases} a_1 \mid a_2 \to a_3 \\ a_3 \bullet a_4 \to a_5 \end{cases} \Rightarrow \quad (a_1 \mid a_2) \bullet a_4 \to a_5$

3. $\begin{cases} a_1 \mid a_4 \mid (a_{67} \bullet (a_{61} \mid a_{79}) \bullet (a_{76} \mid a_{77})) \to a_{100} \\ (a_{45} \bullet a_{47}) \mid (a_{43} \bullet a_{74}) \to a_{67} \bullet (a_{61} \mid a_{79}) \end{cases} \Rightarrow \quad a_1 \mid a_4 \mid (((a_{45} \bullet a_{47}) \mid (a_{43} \bullet a_{74})) \bullet (a_{76} \mid a_{77}))$

Описанные действия осуществляются функцией `DoDSubst()`.

6.3. ψ-перегруппировка

Введение в *R*-алгоритм операции ψ-перегруппировки необходимо для устранения неоднозначностей при записи в виде *R*-выражений фрагментов алгоритмов, содержащих параллельные альтернативы. С т.з. граф-схемы заданного алгоритма параллель-

ные альтернативы ($(a_1 \mid a_3) \bullet (a_2 \mid a_4)$) и альтернативные параллельные ветви ($(a_1 \bullet a_2) \mid (a_3 \bullet a_4)$) – одно и то же, однако с т.з. интерпретации в виде *R*-выражений это не так. Поэтому возникает необходимость в операции, которая позволила бы переходить от одной формы записи к другой.

Алгоритм:

1. Найти выражение *j*, левая часть которого является *ST*-деревом. Если выражение не найдено, перейти к п. 5.
2. Найти выражение *i*, правая часть которого содержит *RT*-поддерево со структурой, соответствующей найденному *ST*-дереву. Если выражение не найдено, перейти к п. 5.
3. Получить перегруппированное подвыражение от *RT*-дерева.
4. Сделать подстановку перегруппированного выражения вместо *RT*-дерева.
5. Конец алгоритма.

Если перегруппировка прошла удачно, то следом за ней необходимо сделать *d*-поглощение для *i*-го и *j*-го выражений.

Примеры ψ-перегруппировки:

1. $\begin{cases} a_0 \to (a_1 \mid a_2) \bullet (a_3 \mid a_4) \bullet (a_5 \mid a_6) \\ a_1 \bullet a_3 \bullet a_5 \to (a_7 \mid a_8) \bullet a_9 \bullet a_{10} \bullet a_{11} \end{cases} \Rightarrow \begin{cases} a_0 \to (a_1 \bullet a_3 \bullet a_5) \mid (a_2 \bullet a_4 \bullet a_6) \\ a_1 \bullet a_3 \bullet a_5 \to (a_7 \mid a_8) \bullet a_9 \bullet a_{10} \bullet a_{11} \end{cases}$
2. $\begin{cases} a_0 \to (a_1 \mid a_2) \bullet (a_3 \mid a_4) \\ (a_1 \bullet a_3) \mid (a_2 \bullet a_4) \to a_7 \bullet a_8 \bullet a_9 \bullet a_{10} \end{cases} \Rightarrow \begin{cases} a_0 \to (a_1 \bullet a_3) \mid (a_2 \bullet a_4) \\ (a_1 \bullet a_3) \mid (a_2 \bullet a_4) \to a_7 \bullet a_8 \bullet a_9 \bullet a_{10} \end{cases}$
3. $\begin{cases} a_0 \to (a_1 \mid a_2 \mid a_3) \bullet (a_4 \mid a_5 \mid a_6) \\ a_1 \bullet a_4 \to a_7 \bullet a_8 \bullet a_9 \end{cases} \Rightarrow \begin{cases} a_0 \to (a_1 \bullet a_4) \mid (a_2 \bullet a_5) \mid (a_3 \bullet a_6) \\ a_1 \bullet a_4 \to a_7 \bullet a_8 \bullet a_9 \end{cases}$

ψ-перегруппировка выполняется функцией `DoPsiRegr()`.

7. РЕАЛИЗАЦИЯ *R*-АЛГОРИТМА ПОИСКА БАЗОВОГО СЕЧЕНИЯ

R-алгоритм заключается в применении операций *u*-, *d*-поглощения и ψ-перегруппировки к исходной системе с целью преобразования ее к т.н. нередуцируемой ω-форме, которая содержит 2 выражения, из которых можно выделить искомое базовое сечение.

Алгоритм применения операций:

1. Найти такие выражения *i* и *j*, для которых применимо правило *u*- или *d*-поглощения. Если такие выражения найдены, то выполнить поглощение, в противном случае перейти к п.3.
2. Если число выражений равно двум, то перейти к п.5
3. Если на данной итерации алгоритма была выполнена операция поглощения, то перейти к п.1.
4. Найти такие выражения *i* и *j*, для которых применимо правило ψ-перегруппировки. Если такие выражения найдены, то выполнить перегруппировку, перейти к п.1, иначе сечение не найдено, перейти к п.6.
5. Выделить из системы базовое сечение.
6. Конец алгоритма.

В ходе выполнения преобразования системы происходит генерация протокола преобразований (последовательность действий) и последующая запись его в файл. Пример преобразования системы по *R*-алгоритму (протокол преобразования):

Исходная система

```
0: a0 -> a1
1: a1 -> a2*a3*a4
2: a3 -> a7|a8|(a9*a10)
3: a4 -> a5*a6
4: a6*(a7|a8|(a9*a10)) -> a11*a12
5: a2*a11 -> a15*a16
6: a12 -> a13|a14
7: a5*a15*(a13|a14) -> a17
8: a16*a17 -> a18
9: a18 -> a19
```

u-поглощение 0, 1:

```
0: a0 -> a2*a3*a4
1: a3 -> a7|a8|(a9*a10)
2: a4 -> a5*a6
3: a6*(a7|a8|(a9*a10)) -> a11*a12
4: a2*a11 -> a15*a16
5: a12 -> a13|a14
6: a5*a15*(a13|a14) -> a17
7: a16*a17 -> a18
8: a18 -> a19
```

u-поглощение 0, 1:

0: a0 -> a2*a4*(a7|a8|(a9*a10))
1: a4 -> a5*a6
2: a6*(a7|a8|(a9*a10)) -> a11*a12
3: a2*a11 -> a15*a16
4: a12 -> a13|a14
5: a5*a15*(a13|a14) -> a17
6: a16*a17 -> a18
7: a18 -> a19

u-поглощение 0, 1:

0: a0 -> a2*a5*a6*(a7|a8|(a9*a10))
1: a6*(a7|a8|(a9*a10)) -> a11*a12
2: a2*a11 -> a15*a16
3: a12 -> a13|a14
4: a5*a15*(a13|a14) -> a17
5: a16*a17 -> a18
6: a18 -> a19

u-поглощение 1, 3:

0: a0 -> a2*a5*a6*(a7|a8|(a9*a10))
1: a6*(a7|a8|(a9*a10)) -> a11*(a13|a14)
2: a2*a11 -> a15*a16
3: a5*a15*(a13|a14) -> a17
4: a16*a17 -> a18
5: a18 -> a19

d-поглощение 3, 4:

0: a0 -> a2*a5*a6*(a7|a8|(a9*a10))
1: a6*(a7|a8|(a9*a10)) -> a11*(a13|a14)
2: a2*a11 -> a15*a16
3: a5*a15*a16*(a13|a14) -> a18
4: a18 -> a19

d-поглощение 2, 3:

0: a0 -> a2*a5*a6*(a7|a8|(a9*a10))
1: a6*(a7|a8|(a9*a10)) -> a11*(a13|a14)
2: a2*a5*a11*(a13|a14) -> a18
3: a18 -> a19

d-поглощение 1, 2:

0: a0 -> a2*a5*a6*(a7|a8|(a9*a10))
1: a2*a5*a6*(a7|a8|(a9*a10)) -> a18

2: a18 -> a19

d-поглощение 1, 2:

0: a0 -> a2*a5*a6*(a7|a8|(a9*a10))

1: a2*a5*a6*(a7|a8|(a9*a10)) -> a19

Базовое сечение:

a2*a5*a6*(a7|a8|(a9*a10))

w-мощность:

5

Поиск базового сечения производится функцией `GetBaseSect()`.

БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1. Зотов И.В., Титов В.С. и др. Организация и синтез микропрограммных мультимикроконтроллеров. – Курск: Курск, 1999. – с.115–164.
2. Баранов С.И., Журавина С.Н., Песчанский В.А. Метод представления параллельных граф-схем алгоритмов совокупностями последовательных граф-схем // А и ВТ. – 1984. – №5. – с. 74–81.
3. Баранов С.И., Журавина С.Н., Песчанский В.А. Обобщенный метод декомпозиции граф-схем алгоритмов // А и ВТ. – 1982. – №5. – с. 43–51
4. Зотов И.В., Колосков В.А., Титов В.С. Выбор оптимальных разбиений алгоритмов при проектировании микроконтроллерных сетей // А и ВТ. – 1997. – №5. – с.51–62
5. Харченко В.С., Кальченко С.Б., Сазонов А.Е. Декомпозиция параллельных матричных схем алгоритмов в задачах синтеза микроконтроллерных сетей // А и ВТ. – 1990. – №4. – с. 81–89
6. Харченко В.С., Никольский С.Б., Сазонов А.Е. Один подход к синтезу дискретных микроконтроллерных сетей // А и ВТ. – 1989. – №4. – с. 87–95