Интеллектуальные системы – 3

Алгоритм Rete (Рити, Реете) (сеть)

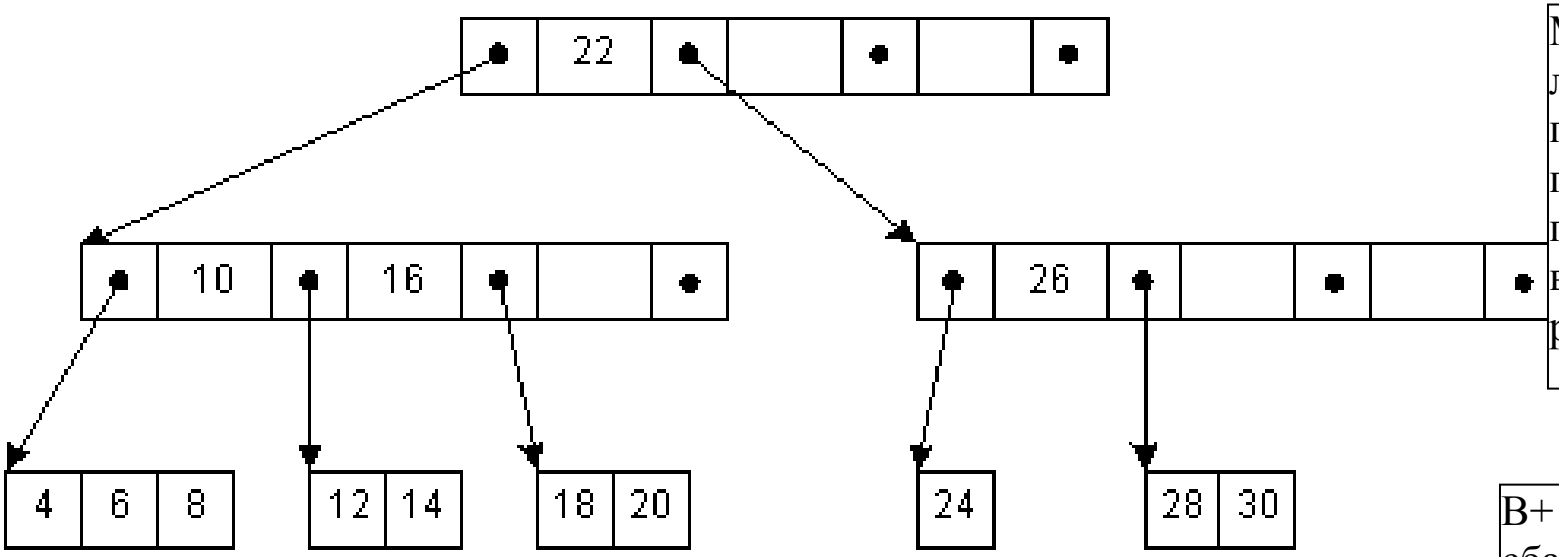
Rete — эффективный алгоритм сопоставления с образцом для продукционных систем , экспертных систем и баз знаний , созданный Чарльзом Форги из Университета Карнеги Меллона. Впервые был описан в рабочем документе 1974 года, затем в докторской диссертации 1979 и в статье 1982 года. Rete стал основой многих популярных экспертных систем, включая CLIPS , Jess , Drools , BizTalk Rules Engine и Soar .

При наивной реализации экспертная система проверяет применимость каждого правила вывода к каждому факту базы знаний, при необходимости выполняет его и переходит к следующему правилу, возвращаясь в начало при исчерпании всех правил. Даже для небольшого набора правил и фактов такой метод работает неприемлемо медленно. Алгоритм Rete обеспечивает более высокую эффективность. При использовании Rete экспертная система строит специальный граф или префиксное дерево, узлам которого соответствуют части условий правил.

Путь от корня до листа образует полное условие некоторой продукции. В процессе работы каждый узел хранит список фактов, соответствующих условию. При добавлении или модификации факта он прогоняется по сети, при этом отмечаются узлы, условиям которых данный факт соответствует. При выполнении полного условия правила, когда система достигает листа графа, правило выполняется.

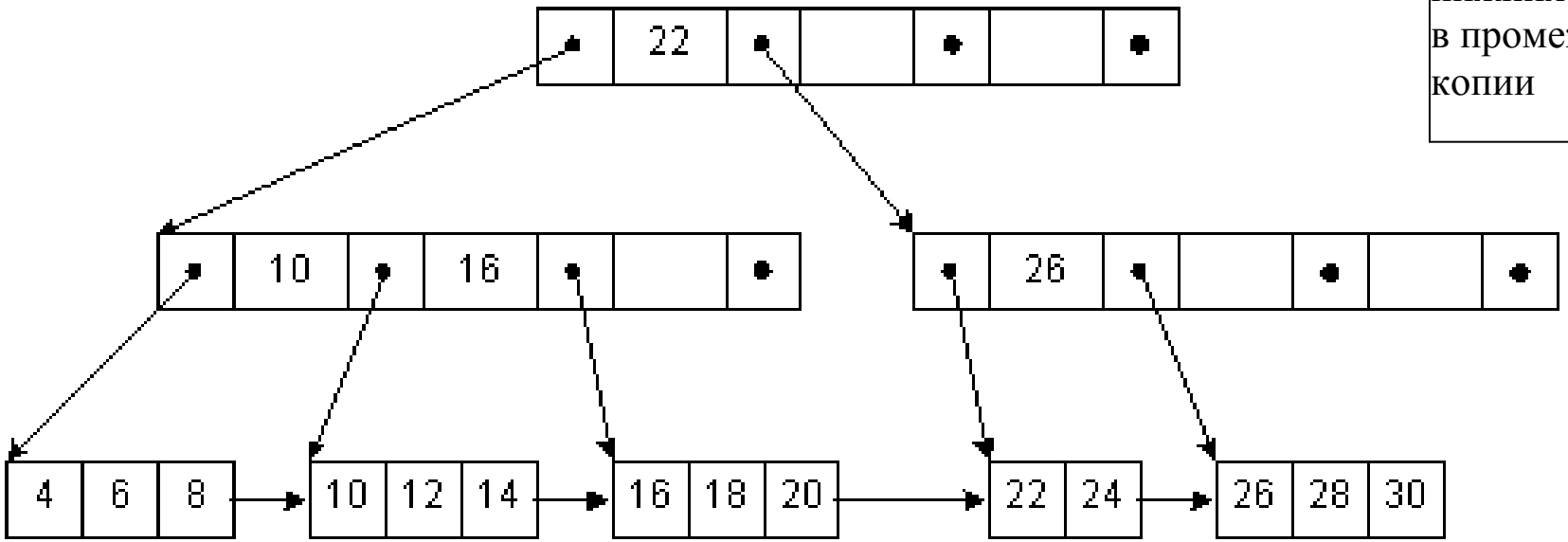
Алгоритм Rete жертвует объёмом памяти ради скорости. В большинстве случаев скорость возрастает на порядки (так как эффективность теоретически не зависит от числа правил в системе). В экспертных системах с большим числом правил классический Rete требует слишком много памяти, но появились новые алгоритмы, в том числе основанные на Rete, ограничивающиеся разумным объёмом памяти

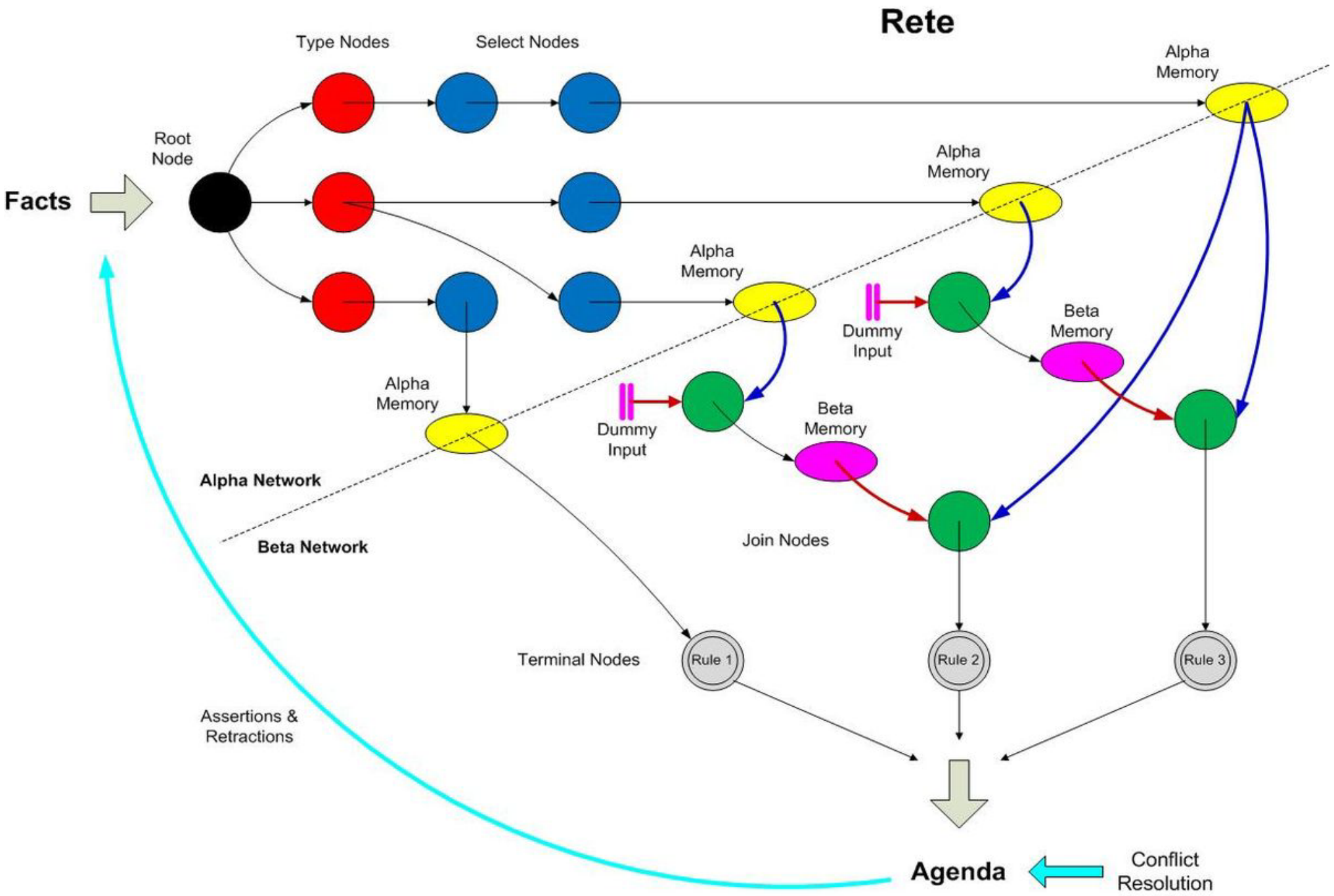
Термин «rete-алгоритм» происходит от латинского слова rete, означающее сеть. Согласно названию, rete-алгоритм функционирует как сеть, предназначенная для хранения большого объема знаний. Он основан на использовании динамической структуры данных, которая автоматически реорганизуется с целью оптимизации поиска аналогично B-дереву, который применяется при индексации структур реляционных баз данных (здесь используется так называемая альфа и бета сети)



Максимальный размер листа не должен превышаться, если превышен, то элемент посередине переносится в верхний узел дерева, а лист разбивается на две части.

B+ дерево, в отличие от B сбалансированного дерева, все данные хранятся в нижних листах,
в промежуточных хранятся копии




Rete
Type Nodes
Select Nodes
Alpha Memory
Root Node
Facts
Alpha Memory
Alpha Memory
Dummy Input
Beta Memory
Alpha Memory
Dummy Input
Beta Memory
Alpha Network
Beta Network
Join Nodes
Terminal Nodes
Rule 1
Rule 2
Rule 3
Assertions & Retractions
Agenda
Conflict Resolution

Итак:

Rete-алгоритм является высокоскоростным средством сравнения фактов с шаблонами, быстродействие которого достигается благодаря хранению в оперативной памяти информации о правилах, которые находятся в сети.

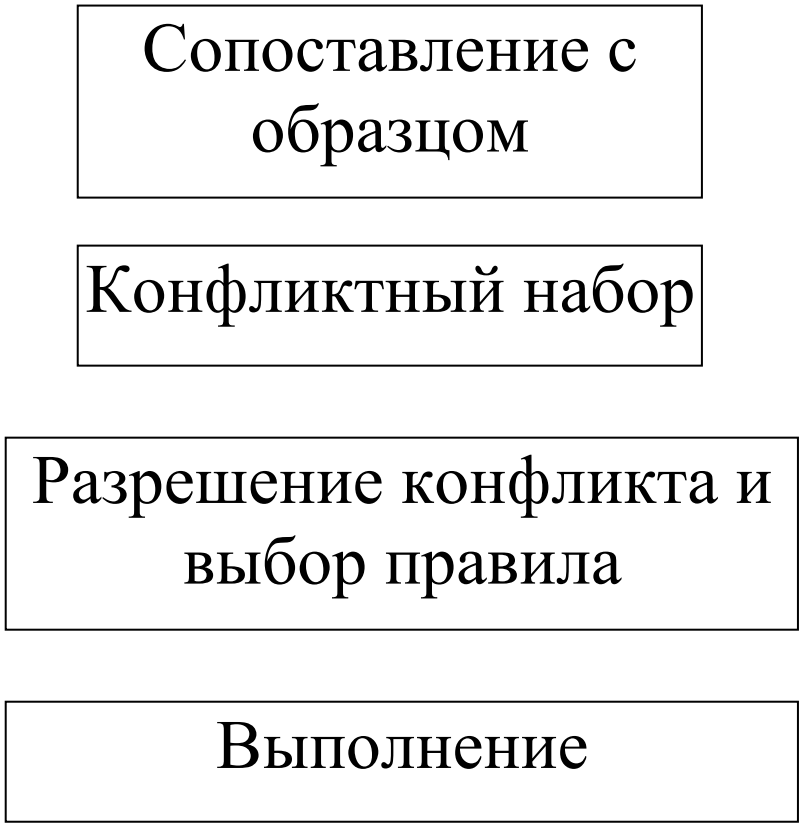
В rete-алгоритме воплощены два эмпирические наблюдения, на основании которых была предложена структура данных, положенных в основу:

Временная избыточность. Действие, оказываемое в результате запуска одного из правил, обычно связана с несколькими фактами.

Структурное сходство. Один и тот же шаблон часто находится в левой части более чем одного правила.

В rete-алгоритме в циклах «распознавание-действие» контролируются только изменения в согласованиях, поэтому в каждом цикле нет необходимости согласовывать факты с каждым правилом. Благодаря этому существенно повышается скорость согласования фактов из антецедентами, поскольку статистические данные, которые не изменяются от цикла к циклу, могут быть проигнорированы.

Типичная продукционная система



Разрешение конфликтов

В цикле сопоставление-решение-действие система находит все возможные сопоставления актуальных фактов. Когда соответствующие продукции поставлены в повестку дня, система определяет порядок выполнения продукций. Список продукций называется конфликтным множеством, а определение порядка их выполнения — разрешением конфликта.

Порядок может быть основан на приоритете, порядке правил, времени актуализации фактов от которых зависят продукции, сложности продукций и на других критериях. Многие системы позволяют разработчику выбрать стратегию разрешения конфликтов или определить очередь из нескольких стратегий.

Разрешение конфликтов тесно связано с алгоритмом Rete, но не является его частью. Некоторые продукционные системы не выполняют разрешение конфликтов.

Rete II

В 1980-х Чарльз Форги создал новую версию алгоритма Rete II. Детали алгоритма не разглашаются. По заявлениям Rete II демонстрирует повышенную эффективность (возможно на порядки) на сложных задачах. Он официально реализован в CLIPS/R2.

Rete II улучшен по двум параметрам: повышена общая производительность сети включая хэшированную память для больших массивов данных, добавлен алгоритм обратного вывода, работающий на той же сети. Jess начиная с версии 5.0 содержит алгоритм обратного вывода по сети Rete, нельзя говорить о полной реализации Rete II, так как полная спецификация последнего не была опубликована.

Rete-NT

В 2010 году др. Ч. Форги разработал новое поколение алгоритма Rete. В бенчмарке журнала InfoWorld алгоритм был признан 500 раз более быстрым, чем оригинальный алгоритм Rete и в 10 раз более быстрым, чем его предшественник Rete II . Этот алгоритм сейчас лицензирован компании «Sparkling Logic», в которую Чарльз вошел как инвестор и консультант по стратегии, для машины вывода продукта SMARTS.

Правило1:

(р **Зеленая_пирамида**

(блок имя **<x>**)

(основание блок **<x>** форма **квадрат** площадь **>1**)

(боковаяп блок **<x>** угол **<90** поверхность **плоскость** цвет **зеленый**)

(верх блок <x> поверхность **точка**)

→ (make (class блок <x> тип Зеленая_пирамида)))

Правило 2:

(р **цилиндр**

(блок имя **<x>**)

(основание блок **<x>** форма **круг** площадь **>1**)

(боковаяп блок **<x>** угол **90** поверхность **кривая**)

(верх блок **<x>** поверхность **плоскость**)

→ (make (class блок <x> тип цилиндр)))

Правило 3:

(р **конус**

(блок имя **<x>**)

(основание блок **<x>** форма **круг** площадь 1)

(боковаяп блок **<x>** угол **<90** поверхность **кривая** цвет **черный**)

(верх блок **<x>** поверхность **точка**)

-> (make (class блок <x> тип Конус)))

Правило 4:

(р купол

(блок имя <x>)

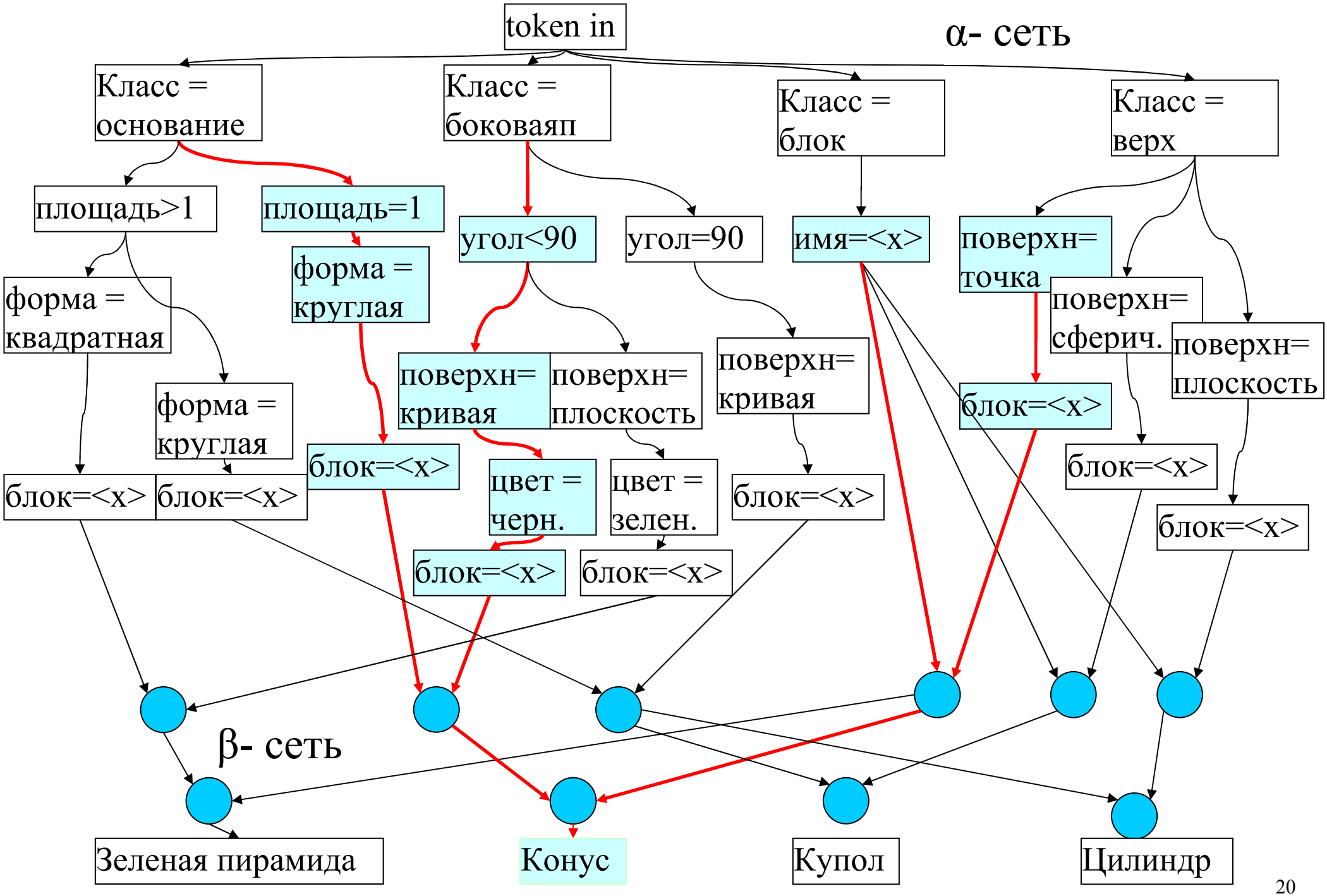
(основание блок <x> форма круг площадь >1)

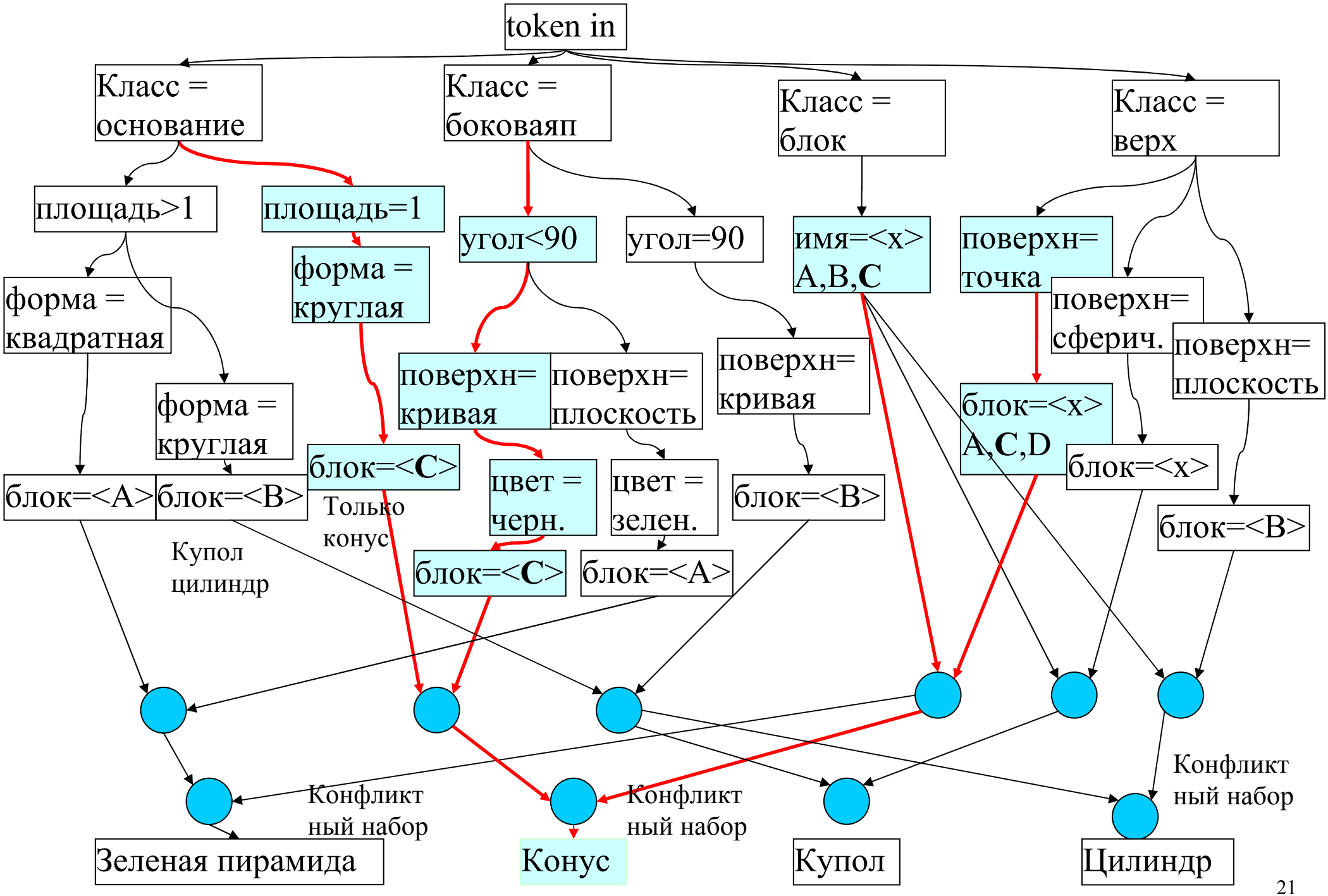
(боковаяп блок <x> угол 90 поверхность кривая)

(верх блок <x> поверхность сферическая)

-> (make (class блок <x> тип купол)))

1. (основание блок A форма квадрат площадь 20)
2. (основание блок B форма круг площадь 20)
3. (основание блок C форма круг площадь 1)
4. (боковаяп блок C угол 85 поверхность кривая цвет черный)
5. (боковаяп блок A угол 45 поверхность плоскость цвет зелен.)
6. (верх блок A поверхность точка)
7. (верх блок C поверхность точка)
8. (верх блок D поверхность точка)
9. (боковаяп блок B угол 90 поверхность кривая)
10. (верх блок B поверхность плоскость)
11. (блок имя A)
12. (блок имя B)
13. (блок имя C)


token in
α- сеть
Класс = основание
Класс = боковаяп
Класс = блок
Класс = верх
площадь>1
площадь=1
угол<90
угол=90
имя=<x>
поверхн= точка
форма = квадратная
форма = круглая
поверхн= сферич.
поверхн= плоскость
поверхн= кривая
поверхн= плоскость
поверхн= кривая
форма = квадратная
форма = круглая
блок=<x>
блок=<x>
блок=<x>
блок=<x>
цвет = черн.
цвет = зелен.
блок=<x>
блок=<x>
блок=<x>
блок=<x>
блок=<x>
β- сеть
Зеленая пирамида
Конус
Купол
Цилиндр


token in
Класс = основание
Класс = боковаяп
Класс = блок
Класс = верх
площадь>1
площадь=1
угол<90
угол=90
имя=<x> A,B,C
поверхн= точка
форма = квадратная
форма = круглая
поверхн= сферич.
поверхн= плоскость
форма = квадратная
форма = круглая
поверхн= кривая
поверхн= плоскость
поверхн= кривая
блок=<x> A,C,D
блок=<x>
блок=<A>
блок=<B>
блок=<C>
Только конус
цвет = черн.
цвет = зелен.
блок=<B>
блок=<B>
Купол цилиндр
блок=<C>
блок=<A>
Конфликт ный набор
Конфликт ный набор
Конфликт ный набор
Зеленая пирамида
Конус
Купол
Цилиндр

Альфа сеть

Левая (альфа) часть графа образует сортировочную сеть , ответственную за выбор записей по соответствию их атрибутов установленным константам. Узел может проверять несколько атрибутов записи. Если запись соответствует условию, она передаётся дальше по графу. В большинстве систем первые от корня узлы проверяют идентификатор или тип записи, поэтому все записи одного типа направляются по одной ветви сортировочной сети.

Каждая ветвь содержит память, в которой накапливаются записи. Записи, не соответствующие хотя бы одному условию, не попадают в соответствующие альфа-списки. Ветви альфа-узлов могут разделяться для уменьшения избыточности условий.

Возможной модификацией является добавление памяти к каждому узлу. Это может быть удобно, когда правила добавляются и удаляются в процессе работы и требуется соответственно модифицировать топологию сети.

Другая реализация описана в Doorenbos . Сортировочная сеть заменена буфером памяти с индексом, возможно в виде хэша . Память содержит записи соответствующие одному шаблону, а индекс указывает в элементы памяти других шаблонов. Данный подход применим только для записей небольшой фиксированной длины. И только для условий, проверяющих значения на равенство. Индекс направляет запись прямо в нужный узел. В такой сети нет узлов с одним входом, проверка условий, отличных от равенства, должна выполняться до индекса, либо такие условия могут быть обработаны в бета-сети.

Бета сеть

Правая (бета) часть графа выполняет сборку записей. Она используется только при необходимости. Каждый узел имеет 2 входа и направляет результат в бета-память.

Бета-узлы работают с метками (tokens), обозначающими записи в альфа-памяти. Метка является единицей хранения и пересылки между узлами и памятью.

Бета-узел в качестве результата может создавать новые метки для списка записей или создавать списки меток.

Обычно создаются списки, где каждая метка отражает одну запись, а набор записей для частичного соответствия представляется списком меток. Этот подход не требует копирования самих записей, узел просто создаёт новую метку, добавляет её к голове списка и сохраняет в буфере выхода.

В описании Rete принято говорить о передаче меток в бета-сети, но в данной статье мы используем записи, так как это более обобщенное представление. При прохождении записи по бета-сети к ней добавляются новые атрибуты. Такая запись в бета-сети представляет собой частичное соответствие условиям некоторой продукции.

Записи в конечных узлах бета-сети полностью соответствуют условию продукции и передаются в выходы Rete-сети, именуемые ‘п-узлы’ от слова ‘продукция’. При попадании записи в п-узел экземпляр правила продукции передается в «повестку дня», где они хранятся в списке с приоритетом .

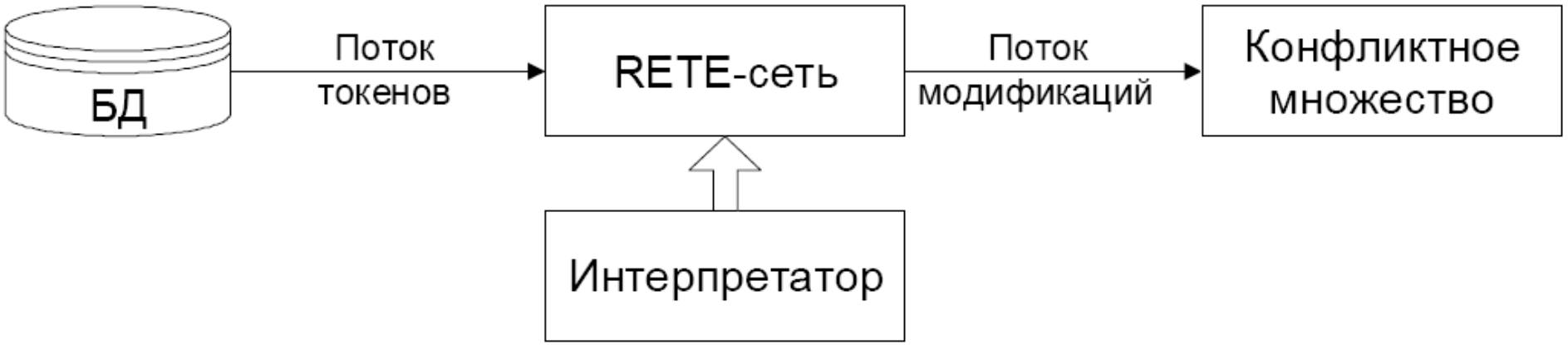
Бета-узлы объединяют списки записей бета-памяти и отдельных записей альфа-памяти. Каждый бета-узел связан с двумя входными блоками памяти. Альфа-память хранит записи и активирует бета-узлы ‘справа’ при поступлении каждой новой записи. Бета-память хранит списки записей и при поступлении записей активирует бета-узлы ‘слева’. При правой активации объединяющий узел сравнивает один или несколько атрибутов добавленной записи из входной альфа-памяти с соответствующими атрибутами каждой записи входной бета-памяти. При левой активации он просматривает добавленный список записей в бета-памяти, извлекает значения нужных атрибутов и сравнивает эти значения со значениями атрибутов каждой записи в альфа-памяти.

Списки записей на выходе из бета-узлов попадают либо в бета-память либо передаются напрямую в конечные узлы. Списки сохраняются в бета-памяти независимо от решения выполнить левую активацию следующих бета-узлов.

Бета-узлы в последнем слое не имеют входов от вышестоящих бета-узлов. В одних системах используют специальные узлы-адаптеры для связи альфа-памяти и левого входа бета-узлов. В других бета-узлы связаны напрямую из альфа-памяти, считая это левым или правым входом. В любом случае оба входа конечных узлов связаны с альфа-памятью.

Для устранения избыточности узлов любая альфа- или бета-память может активировать множество бета-узлов. Как и с объединяющими узлами в бета-сети могут быть дополнительные типы узлов, некоторые из которых описаны ниже. Если Rete сеть не содержит бета-сети, альфа-узлы выдают метки, содержащие по одной записи, напрямую в п-узлы. В этом случае нет необходимости хранить записи в альфа-памяти.

RETE-сеть представляет собой потоковую структуру данных. На её вход поступает поток токенов, отражающих изменения в БД. На выходе формируется поток модификаций в конфликтном множестве.



База данных состоит из элементов рабочей памяти (ЭРП) ПС. Продукция может быть представлена таким образом, что каждый ее условный элемент (УЭ) имеет вид элементарной четвёрки <класс, объект, атрибут, значение>. В соответствии с этим ЭРП принимают аналогичный унифицированный вид.

Формальная запись:

WME = <Cls, Obj, Attr, Val>,

где WME – элемент рабочей памяти, Cls – класс объекта, Obj – идентификатор объекта, Attr – атрибут, Val – значение.

Добавление или удаление ЭРП в базе данных представляется токеном, поступающим на вход RETE-сети. Входной токен состоит из двух компонентов – тэга и ЭРП. Тэг передаёт знак токена. Положительным считается токен, соответствующий добавлению ЭРП, отрицательным – удалению. Таким образом, формально можно записать

$$TKNin = \langle TG, WME \rangle,$$

где TKNin – входной токен, TG – тэг, WME – ЭРП.

Конфликтное множество представляет собой набор упорядоченных пар, каждая из которых содержит продукцию и список удовлетворяющих ей объектов:

$$CS = < (PR1 , OL1), (PR2, OL2), (PR1, OL1), ..., (PRP, OL P) >,$$

где CS – конфликтное множество;

PR1 ... PR P – продукции;

OL1 ... OLP – списки удовлетворяющих им объектов;

P – количество продукций.

При появлении в БД набора объектов, свойства которых удовлетворяют условию продукции, соответствующий набор вносится в конфликтное множество.

Выходной токен RETE-сети содержит пару (PRi, OLi) конфликтного множества.

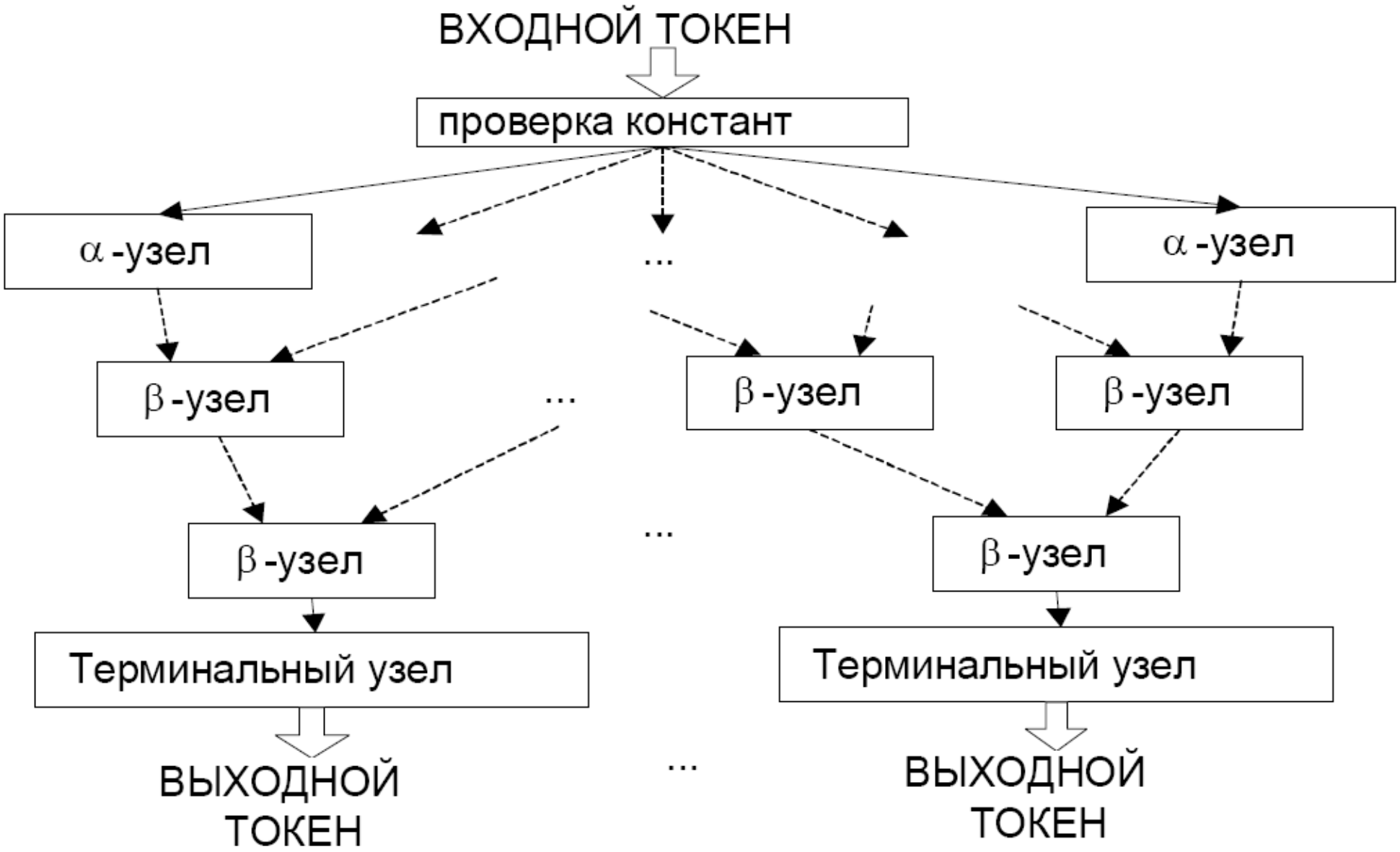
Операции над конфликтным множеством можно формально записать следующим образом:

$CS_{t+1} = CS_t \cup \{(PRi, OLi)\}$ – при положительном тэге токена,

$CS_{t+1} = CS_t \setminus \{(PRi, OLi)\}$ – при отрицательном тэге токена,

где t – время до итерации;

t+1 – время после итерации.



Структура Rete сети – граф потока данных

Подсеть проверки констант выполняет роль сортировщика входных токенов и имеет один вход и множество выходов. Каждому возможному сочетанию констант соответствует отдельный выход.

α-узел имеет один вход и один выход и предназначен для хранения копии входного токена с определённым набором констант ЭРП. В соответствии с функциональным назначением α-узел содержит память для хранения поступающих токенов. Объём этой памяти равен

$$M_{\alpha} = N_{TKN\alpha} * M_{TKN} ,$$

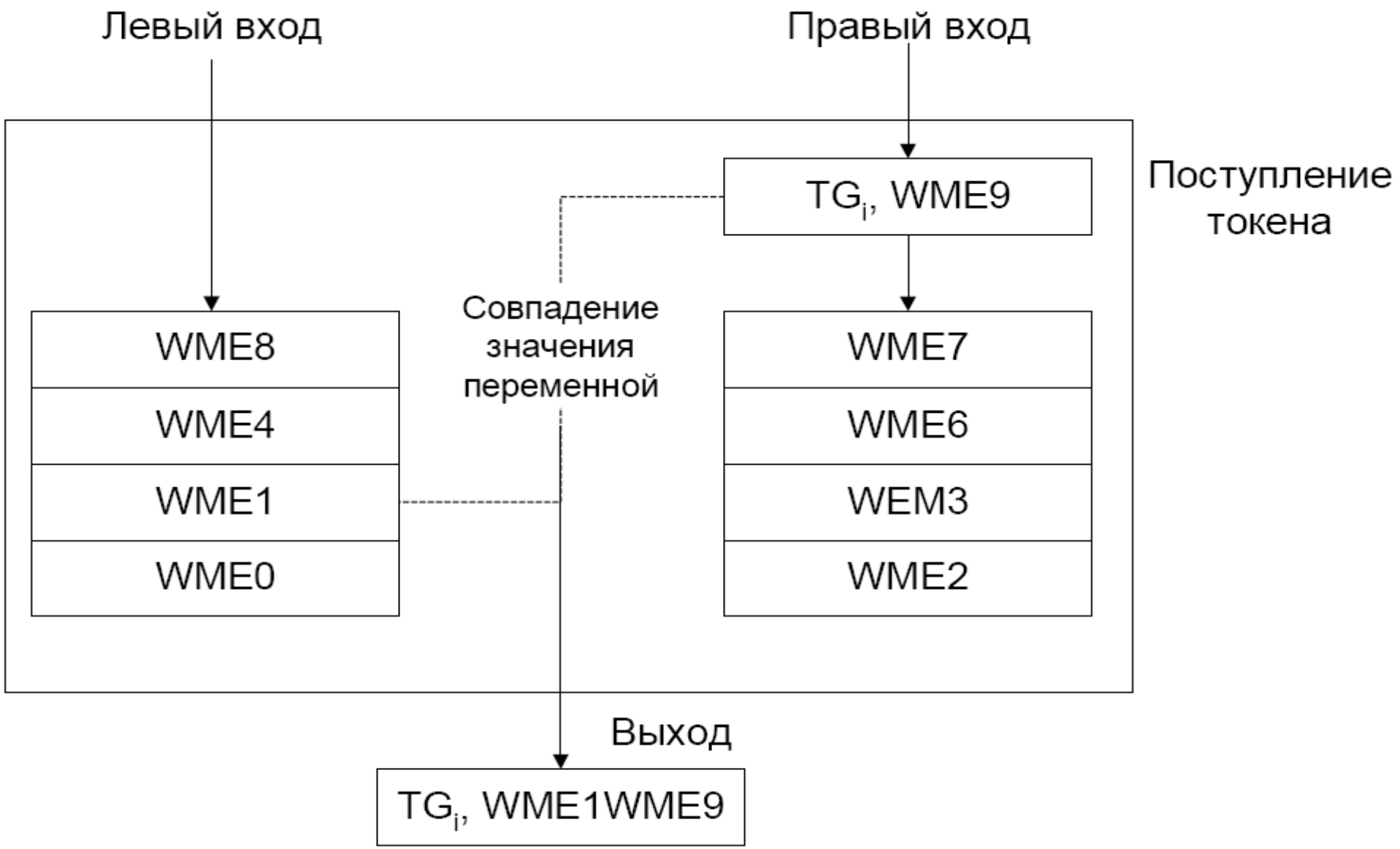
где M_{α} – объем памяти α -узла, $N_{TKN\alpha}$ – количество токенов, подаваемых на вход данного узла, M_{TKN} – объём памяти, занимаемый токеном.

Очевидно, что число токенов, поступающих на каждый α-узел, заведомо неизвестно. При получении оценок целесообразно рассматривать наихудший случай, когда все объекты БД имеют ЭРП с одним и тем же набором констант. Тогда память α -узла должна иметь объём

$$M_{\alpha} = N_{Obj} * M_{TKN} ,$$

где N_{Obj} – количество объектов в БД.

β-узел имеет два или более входов и служит для сопоставления токенов, подаваемых на его входы, на предмет совпадения значений переменной в ЭРП. Если среди токенов, поступивших на разные входы, найдены совпадения значений переменных, то на выходе β-узла формируется токен, состоящий из соответствующей пары (тройки и т.д.) входных токенов. Выходной токен содержит тот же тэг, что и входной. Если на вход поступил положительный токен, то он добавляется к списку соответствующего входа, если отрицательный, то удаляется.



Формирование выходного токена в β-узле

Размер токена определяется количеством содержащихся в нём ЭРП (или ссылок на них). Легко видеть, что размер токена растёт от слоя к слою по мере его продвижения по сети. В терминальном узле соответственно размер токена определяется количеством α-узлов Nα. Объём памяти, занимаемый токеном, поступающим в терминальный узел, равен

$$M_{TKNt} = N\alpha * M_{WME} ,$$

где M_{WME} – объём памяти ЭРП.

Промежуточный токен в RETE-сети фактически является результатом частичного сопоставления и формируется как результат обработки β-подсети, завершающейся соответствующим β -узлом:

$$M_{TKN\beta\_} = N_{\beta in} * M_{WME}$$

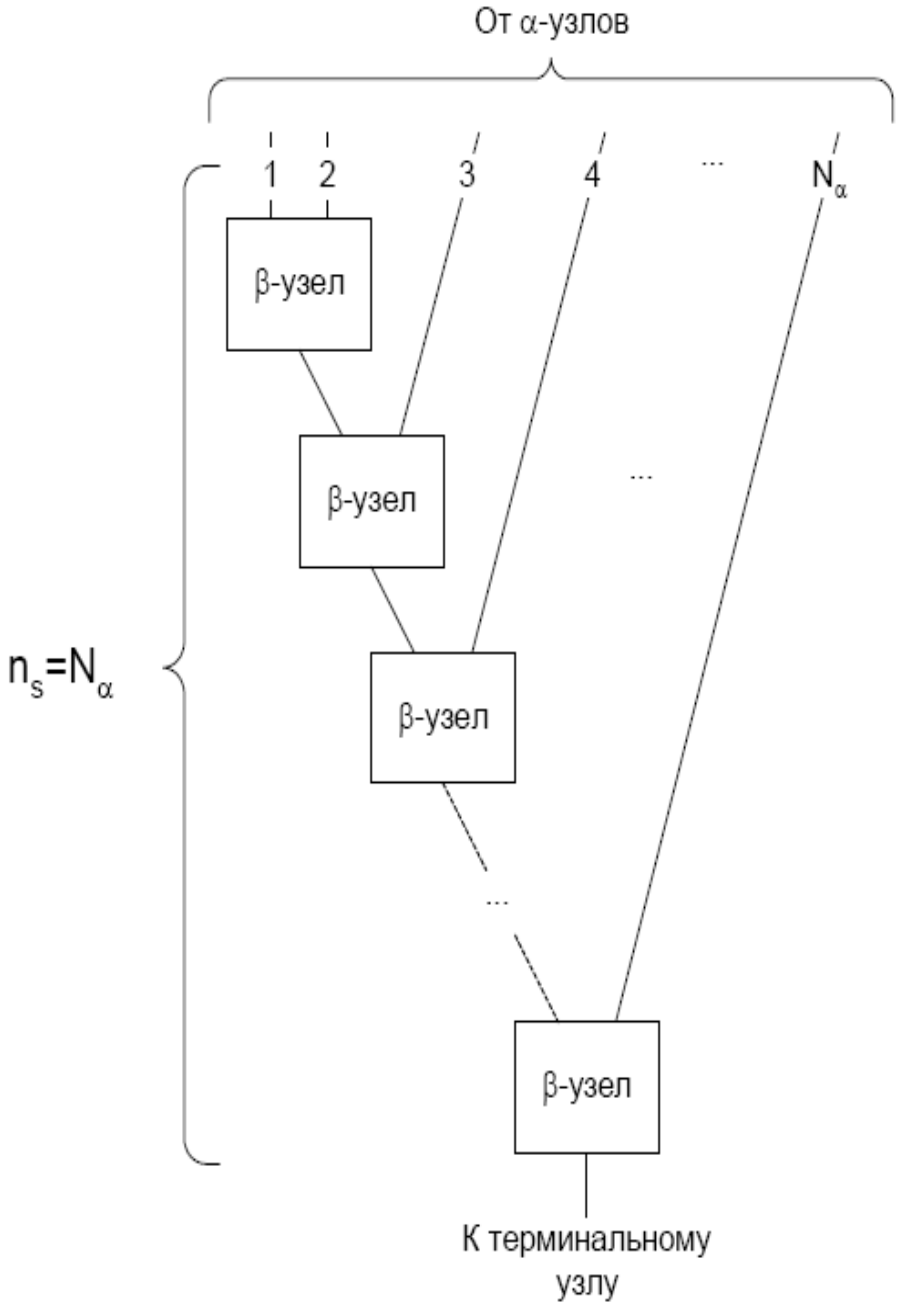
где $M_{TKN\beta}$ – объём памяти, занимаемый промежуточным токеном на выходе β -узла, $N_{\beta in}$ – количество входов β -подсети, которая завершается данным β -узлом.

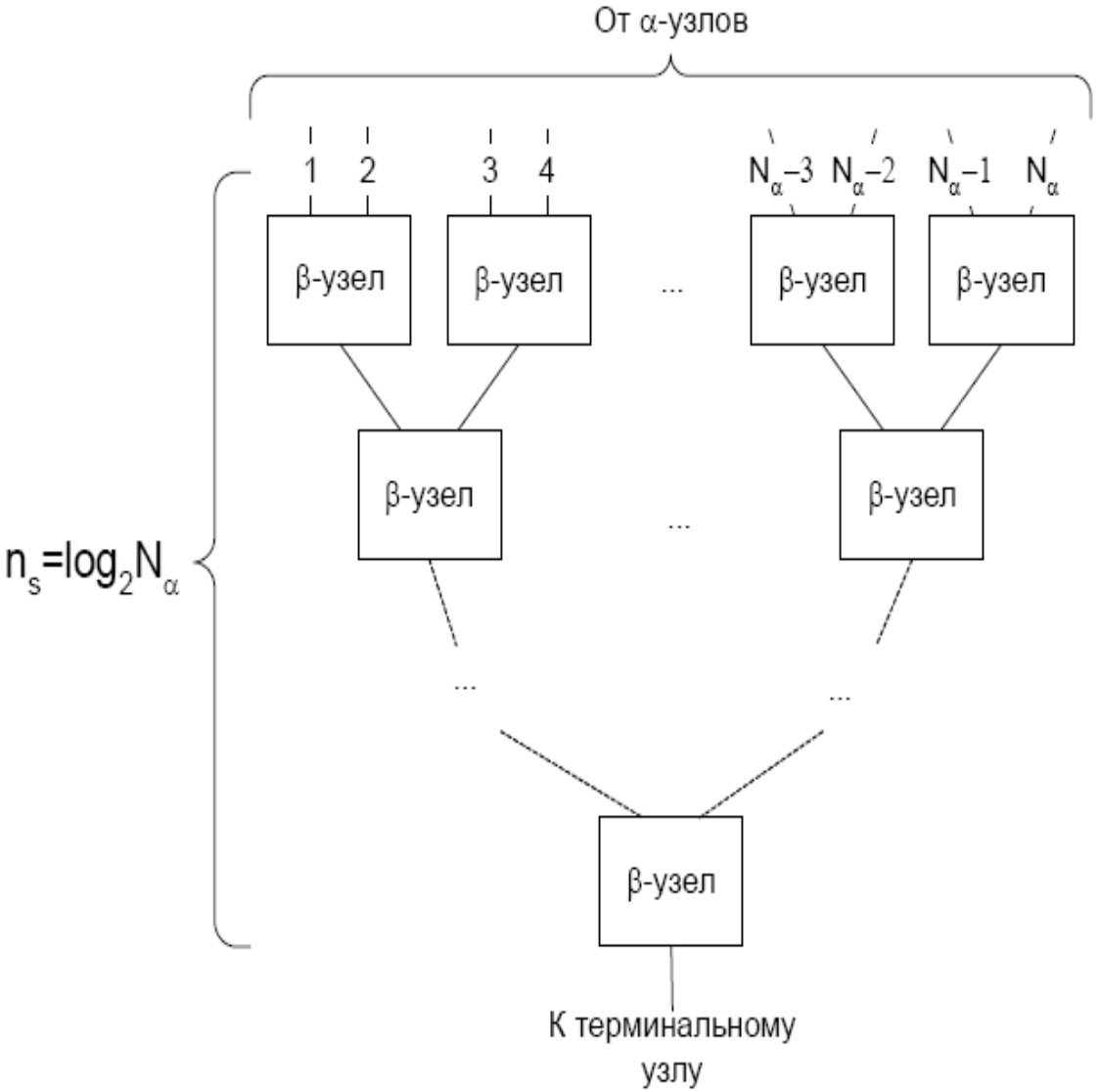
Терминальный узел хранит решения по продукции. Количество терминальных узлов равно числу продукций в ПС. На вход терминального узла поступает промежуточный токен, содержащий полный набор ЭРП, удовлетворяющий условию продукции. На выходе формируется пара (PRi, OLj), где i – идентификатор терминального узла (продукции).

При поступлении входного токена интерпретатор RETE-сети выполняет обход графа с добавлением или удалением соответствующих промежуточных токенов в β-памяти. Если поступивший в БД факт явился завершающим звеном в наборе фактов, удовлетворяющих условию одной из продукций, то обработка соответствующего токена завершается выдачей решения в конфликтное множество.

Производительность ФОП определяется количеством входных токенов, обрабатываемых в единицу времени, которое можно считать пропускной способностью интерпретатора.

Существуют два базовых способа построения структуры β-подсети – в глубину и в ширину. Второй случай очевидно выгоднее первого по величине t_{TKN}, так как количество слоёв n_S существенно меньше. С другой стороны, он более сложен для задачи построения сети, так как количество входов подсети $N\alpha$ не обязательно кратно степени 2 и количество входов β-узла может быть более двух.


От α-узлов
1
2
3
4
...
N_α
β-узел
β-узел
β-узел
β-узел
$n_s=N_\alpha$
К терминальному узлу


От α-узлов
1
2
3
4
$N_\alpha-3$
$N_\alpha-2$
$N_\alpha-1$
N_α
β-узел
β-узел
β-узел
β-узел
β-узел
β-узел
β-узел
$n_s=\log_2 N_\alpha$
К терминальному узлу