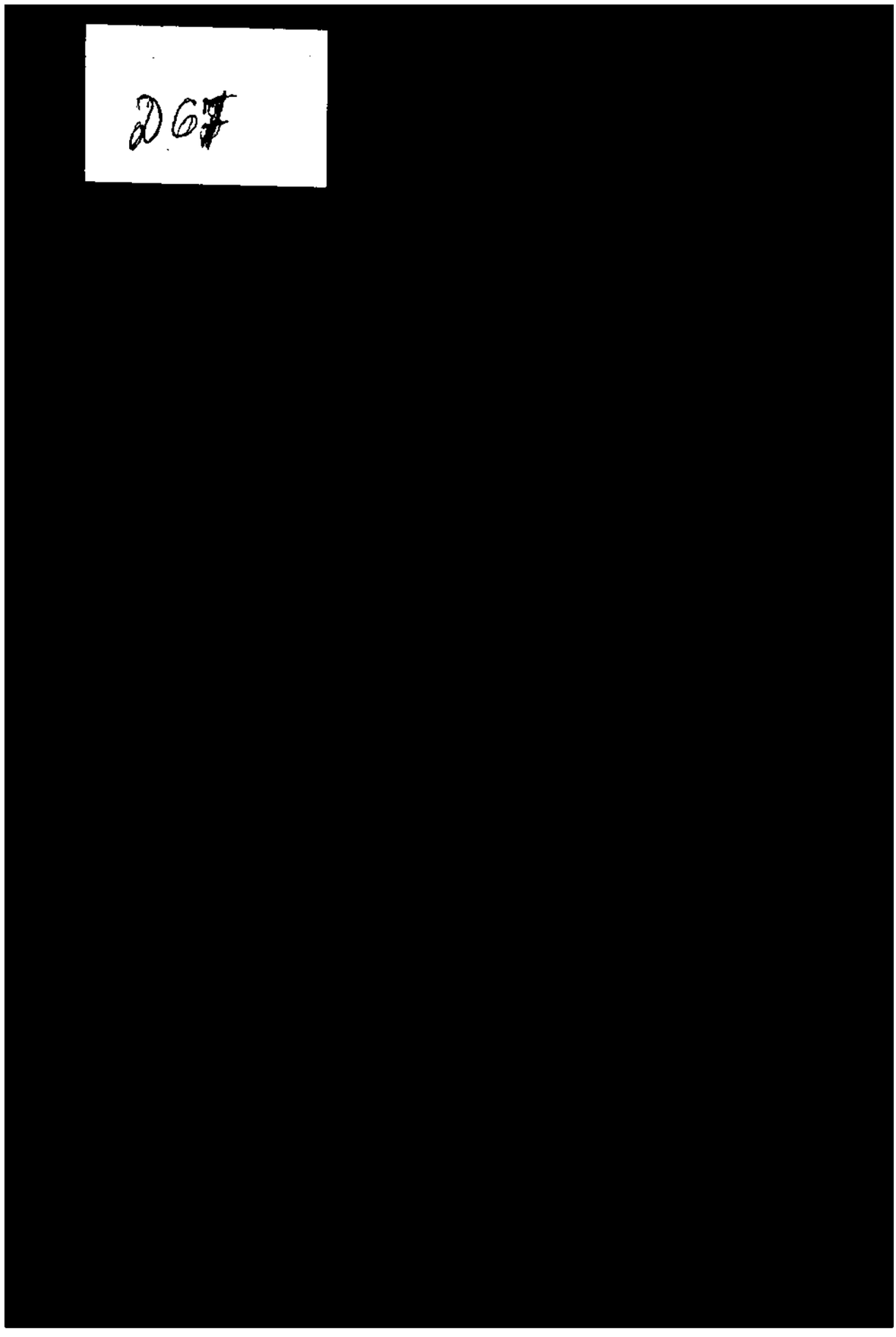

D67

Б Ъ Л Г А Р С К А А К А Д Е М И Я Н А Н А У К И Т Е

ИНСТИТУТ ПО МАТЕМАТИКА С ИЗЧИСЛИТЕЛЕН ЦЕНТЪР

Здравко Иванов Марков

ЕЗИК ЗА РАЗПРЕДЕЛЕНО ЛОГИЧЕСКО ПРОГРАМИРАНЕ

Д И С Е Р Т А Ц И Я

за присъждане на научна степен

"кандидат на математическите науки"

София, 1990 г.

СЪДЪРЖАНИЕ

УВОД.....1

ГЛАВА 1. Паралелно логическо програмиране.....4

1.1. Логическо програмиране и Пролог.....4

1.2. Подходи към паралелна реализация на Пролог.....8

1.3. Резюме.....15

ГЛАВА 2. Език на мрежовите клаузи.....16

2.1. Увод.....16

2.2. Основни символи.....17

2.3. Мрежови клаузи.....17

2.4. Активиране на мрежовите клаузи.....19

2.5. Активиране на процедури във възлите.....21

2.5.1. Разпространяваща се активация.....21

2.5.2. Активация по необходимост.....23

2.5.3. Активация при неуспешна унификация.....26

2.6. Пропадане на процедурите.....28

2.7. Динамична промяна на структурата на мрежата.....32

2.8. Унификация с отлагане.....36

2.9. Резюме.....43

ГЛАВА 3. Логически извод чрез мрежови клаузи.....45

3.1. Извод, воден от данните.....46

3.2. Извод при недетерминирани логически програми.....51

3.3. Логическа семантика на разпространяващата се активация.....56

3.4. Смесена стратегия за извод.....61

3.5. Резюме.....64

ГЛАВА 4. Разсъждения по премълчаване.....65

ГЛАВА 5. Представяне и обработка на многостени.....74

5.1. Списъчно представяне на многостени.....74

5.2. Представяне на многостени чрез базата от данни на Пролог....77

5.3. Представяне на многостени чрез мрежови клаузи.....78

5.3.1. Използуване на разпространяваща се активация.....78

5.3.2. Използуване на активация при неуспешна унификация...82

5.4. Език на мрежовите клаузи и конекционизъм.....83

5.5. Резюме.....87

ГЛАВА 6. Обучение.....88

6.1. Обучение чрез Пролог.....91

6.2. Обучение на мрежови клаузи.....93

6.3. Индуктивно обучение на понятия.....98

6.4. Резюме.....102

ГЛАВА 7. Реализация на езика на мрежовите клаузи.....104

ЗАКЛЮЧЕНИЕ.....106

ЛИТЕРАТУРА.....109

ПУБЛИКАЦИИ ПО ТЕМАТА НА ДИСЕРТАЦИЯТА.....114

УВОД

Разпределената обработка на информацията е класическа област на информатиката, свързана със създаването и използуването на непоследователни (не фон-ноймановн) изчислителни архитектури. Това направление е предмет на изследвне и в рамките на изкуствения интелект (ИИ), едновременно като формализъм, и като инструмент за моделиране на някои когнитивни аспекти на естествения интелект. Типични за този подход в ИИ са изследванията в областта на *невронните мрежи*, започнали с основополагащата работа на Мак Калок и Питс [42]. Тези изследвания донесоха голямата популярност на ИИ през 50-те началото на 60-те години (успехите на перцептрона [59]), както и следващите ги разочарования (изследванията на Мински и Пейперт [46]), довели до появата на чисто *символния* подход към изкуствения интелект (различните методи за представяне и обработка на знания - фреймн, правила и др.). Понастоящем се наблюдава отново нарастващ интерес към невронните мрежи, като основните идеи, които ги възраждат, са новите им приложения и интегрирането им със символния подход. В резултат на това напоследък се обособява една нова дисциплина, наречена в англоезичната литература Connectionism или PDP (Parallel Distributed Processing) [19,3,51,60,61]. Тук ще използуваме термина *разпределена обработка на информацията* или *конекционизъм*.

Първоначално символният и конекционисткият подход се противопоставяха, като изследователите, застъпващи всяка отделна страна, твърдяха че всичко може да се постигне само с единия от подходите. Напоследък се забелязва тенденция за търсене на пътища за съвместното им използуване или интегрирането им. Това е естествен резултат от неуспеха на едностранните подходи при решаването на проблемите на ИИ. Добър анализ в това отношение се прави в [14], където се заключава, че *интелектът не е продукт на нито един конкретен механизъм за представяне и обработка на информация, по-скоро той е резултат от взаимодействието на различни механизми и представяния на различни нива на описание на проблемите*.

Съвременният конекционизъм използува всички достижения на класическите невронни мрежи, като ги развива в две направления. Първото е изследването на нови архитектури и алгоритми за обучение и тяхното прилагане в съвременни области, като например експертните системи. Второто направление се опитва да обедини конекционизма със символния подход в ИИ. *Настоящото изследване се разглежда като стъпка във второто направление.* Особеностите на конекционизма са разгледани по-подробно в раздел 5.4.

Символният подход към ИИ, на който се основава нашата трактовка на разпределената обработка на информацията, е *логическото програмиране* [36]. Това е мощен формализъм с ясна и коректна теоретична основа - предикатното смятане от първи ред. Тази основа прави логическото програмиране удобно за описание на широк клас задачи от областта на символната обработка на информацията и ИИ. То обаче не е само мощен спецификационен език, но и ефективен инструмент за решаване на задачи. Тези две страни на логическото програмиране се съчетават в езика Пролог [12,1], който е основен представител на различните опити за реализация на спецификационния език на логическото програмиране. Те са отразени в двойствената семантика на Пролог - декларативна (спецификационна) и процедурна (свързана с последователната алгоритмична реализация на езика).

Концепцията на Пролог, както и повечето негови реализации се основават на последователни алгоритми, предназначени за класически (последователни) изчислителни архитектури. Въпреки това се правят опити за използуване на паралелни схеми за неговата реализация, като целта в тези случаи е повишаване на ефективността на програмите. Преглед на постигнатото в тази област е направен в глава 1. Важно е да се отбележи обаче, че всички паралелни реализации на Пролог разширяват управляващия език, с което отдалечават езика от неговата логическа основа. В настоящата работа се предлага друг подход към решаването на тази задача. Изходна точка на работата е реализацията на език за разпределено изчисление (*език на мрежовите клаузи*), който в една от своите интерпретации се разглежда като език за логическо програмиране. Реализацията

на дедуктивна система не е основен мотив за създаването на езика, а по-скоро едно от неговите приложения (семантики).

Езикът на мрежовите клаузи е език за разпределена обработка на информацията. Често разпределената обработка се свързва с паралелно изчисление. В настоящото изследване обаче ударението е поставено не върху паралелността на реализацията, свързана с паралелни архитектури (в повечето случаи обаче моделирани на последователни), а върху разпределения характер на информационните обработки. В случая се използува концепцията за разпределената обработка при конекционизма - множество самостоятелни обработващи елементи, свързани в мрежа, при отсъствие на централизирано управление. Важно е да се отбележи, че конекционизмът в случая се разглежда преди всичко като парадигма за организация на изчислителния процес, без да се следват буквално някои от неговите класически особености. Отличителна черта на настоящия подход в това отношение е, че обработващите елементи са сложни и извършват предимно символна обработка, за разлика от класическите формални неврони, реализиращи прости прагови функции. Тази особености определят същността на настоящата работа като *стъпка към интегрирането на конекционизма със символния подход в ИИ.*

БЛАГОДАРНОСТИ

Преди всичко искам да благодаря на Христо Дичев, с който обсъждах почти всички въпроси, засягащи работата, и винаги получавах конструктивна критика, ценни съвети и мотивация за по-нататъшната работа. Благодарен съм и на Лидия Синапова, която първа използува езика на мрежовите клаузи, и със своя интерес и оригинални разработки ме убеди в практическата ценност на езика. Съдействие и помощ получавах от проф. В. Сгурев, които постоянно ме убеждаваше в необходимостта от писане на дисертация, а когато я започнах ме окуражаваше и подкрепяше. Благодарен съм също на ст.н.с. Аврам Ескенази, без чиято ценна помощ и съвети настоящата работа не би достигнала заключителния си етап.

ГЛАВА 1. Паралелно логическо програмиране

В настоящия раздел се въвеждат основните понятия на логическото програмиране и се прави преглед на направленията за развитие на езика Пролог в областта на паралелното програмиране. Разглеждат се проблемите, възникващи при този подход, част от които имат пряка връзка със създаването на езика на мрежовите клаузи.

1.1. Логическо програмиране и Пролог

Логическото програмиране започва своето развитие в началото на 70-те години като наследник на по-ранните идеи за автоматично доказателство на теореми. Основополагаща в това отношение е работата на Робинсон [58], в която той въвежда *правилото на резолюцията* - формализирана процедура за логически извод, удобна за компютърна реализация. Резолюцията е в основата на логическото програмиране, което може също да се разглежда като схема за логически извод. Важно обаче е да се подчертае, че идеята, която роди логическото програмиране и го направи популярно, не е логическата му основа. Тази идея в най-кратка форма е формулирана от Д. Уорън [78]: *логическото програмиране* е *програмиране*. Използването на логиката като език за програмиране, беше задачата, която си поставиха Ковалски [35] и Колмерауер [13]. Решението на тази задача доведе до появата през 1972 г. на езика Пролог - първият и основен представител на езиците за логическо програмиране. Преди това логиката се е използувала в информатиката само като спецификационен език. За да бъде използувано като език за програмиране беше избрано едно ограничено подмножество на логиката - *предикатното смятане от първи ред*, също ограничено до *клаузна форма*. Последното ограничение обаче не е съществено, тъй като то не намалява общността на програмите (съществува алгоритъм за превеждане на всяка формула от първи ред в еквивалентна формула в клаузна форма).

Езикът Пролог доби голяма популярност след издаването на "Програмиране на Пролог" от Клоксин и Мелиш [12] (може би най-добрия учебник по Пролог, издаван 3 пъти), книгите на Стерлинг и Шапиро [68], Братко [7], Кларк и Мак Кейб [11] и много други, които наред с описанието на езика дават и конкретни примери за неговото използуване при решаването на задачи на ИИ.

Логическото програмиране използува синтаксиса на *клаузите на Хорн* [36]. Това са клаузи (дизюнкции на литерали), съдържащи най-много един положителен литерал. Програмите на логическото програмиране се състоят от три вида клаузи на Хорн (в скоби е дадено наименованието и синтаксиса им в Пролог):

1. *Програмни клаузи* (*правила*)

A <-- $B_1, B_2, \ldots, B_n$ (A:-$B_1, B_2, \ldots, B_n$.),

където A се нарича *глава*, а $B_1, B_2, \ldots, B_n$ - *тяло* на клаузата.

2. *Единични клаузи* (*факти*)

B <-- (B.)

3. *Целеви клаузи* (*въпроси* или *цели*)

<-- C (?- C.)

A, B_i ($i=1,\ldots,n$) и C са *литерали*. Литералът е *атомна формула* или отрицание на атомна формула, т.е. израз от вида $p(t_1, t_2, \ldots, t_m)$, където p е *предикатен символ*, а t_i - терми. *Термът* е атом, променлива или структура. *Структурата* е израз от вида $f(t_1, t_2, \ldots, t_m)$, където f е функционален символ а t_i - терми.

Съществуват няколко интерпретации (семантики) на логическите програми, които са свързани с реализацията им или с областта, в която се използуват.

▪ Декларативна интерпретация. Програмната клауза дефинира правилото: A е

вярно ако всичките $B_1, B_2, \ldots, B_n$ са верни. Единичните клаузи дефинират безусловно верни твърдения, а целевата клауза е твърдение, чиято вярност се оценява при наличието на текущите програмни и единични клаузи в програмата.

▪ Процедурна интерпретация. Тази интерпретация лежи в основата на реализацията на Пролог. Тя използува следното съответствие: програмната клауза A се разглежда като дефиниция на процедура, в която целите B_i са обръщения към други процедури. Програмата започва работа от началната цел (въпроса). Ако текущата цел е <-- $C_1, \ldots, C_k$ на всяка стъпка се избира някое C_i, което се унифицира с главата на някоя програмна клауза A <-- $B_1, \ldots, B_n$. По такъв начин се получава нова текуща цел <-- $(C_1, \ldots, C_{i-1}, B_1, \ldots, B_n, \ldots, C_k)s$, където s е субституцията, получена при унификацията на C_i с A. Изпълнението на програмата завършва при достигане на празната цел. В тази схема процедурата за унификация играе ролята на универсален механизъм за предаване на параметри между процедури, достъп и конструиране на данни, а ролята на данните се изпълнява от термите - аргументи на процедурите.

Наличието на декларативна и процедурна интерпретация е важно свойство на логическите програми, което означава, че може да се отдели семантиката на една програма (отговорът на въпроса "какво прави програмата") от изпълнението ѝ ("как се достига резултатът от програмата"). Съществено е да се отбележи, че управляващият компонент на програмата (изпълнението ѝ) не се задава в явен вид от програмиста. Това е описаната по-горе вградената процедура в Пролог, реализираща вариант на метода на резолюцията - т.нар. SLD-резолюция. В термините на резолюцията литералът C_i и главата на клаузата A са два *контрарни литерала* (целта и литералите в тялото на клаузата са отрицателни, а главата на клаузата и фактите - положителни), които се елиминират на всяка стъпка на резолюцията (*резолвиране* на клаузите <-- $C_1, \ldots, C_k$ и A <-- $B_1, \ldots, B_n$).

Наред с вградения механизъм на резолюцията съществуват някои елементи от управлението на логическите програми, които се задават явно. Това са *изборът на цел* C_i, (computation rule) и *изборът на клауза* A (search rule). Тези две правила при Пролог са фиксирани - изборът на цел става от ляво на дясно в

конюнкцията, а изборът на клауза - от горе на долу в базата от данни. Така че начинът на записване на програмата е форма на използуване на тези два управляващи компонента. Правилата за избор на цел и клауза определят до голяма степен адекватността на реализацията на конкретната схема за логическо програмиране по отношение на абстрактния модел на резолюцията. Например стратегията, използувана от Пролог, *не е пълна*, тъй като при рекурсивни програми фиксираният избор на цел и клауза може да доведе до безкраен цикъл, който да блокира пътя към откриване на решението, дори когато то съществува. Правилото за избор на клауза е свързано с една важна особеност на логическите програми - наличието на *недетерминизъм*, т.е. множественост на решенията.

Процедурната семантика на логическите програми може да се опише и с търсене в дърво с два типа възли - И и ИЛИ, т.нар. *И-ИЛИ-дърво*. И-възлите описват правилото за избор на цел, а ИЛИ - правилото за избор на клауза.

Необходимостта от използуване на описаните управляващи правила, както и на вградения предикат за отсичане (!) е съществен недостатък на съществуващите реализации на логическото програмиране. Този проблем, (известен като *проблем на управлението*) заедно с проблема за *коректната реализация на отрицанието* са две трудни задачи, които стоят за решаване при реализацията на логическото програмиране. Те са решени само частично в Пролог, като стремежът е бил да се запази максимално "чистата" декларативна семантика на езика. В паралелните версии на логическото програмиране обаче управляващите компоненти са концептуална основа на техните реализации, така че те заемат основно място в езика, като дори са разширени с допълнителни елементи, управляващи синхронизацията между изпълняваните в паралел цели или клаузи.

• Семантика на база от данни. Логическите програми могат да се разглеждат като база от данни [55]. В този смисъл логическите програми, чиито единични клаузи се състоят само от основни терми (терми не съдържащи променливи) представляват естествено обобщение на релационните бази от данни. Логиката в случая играе ролята на универсален език за представяне на данни, програми, заявки и различни обработки върху данните.

▪ Интерпретация в термините на процеси. Една конюнкция от цели се разглежда като система от конкурентни (паралелни) процеси. На всяка стъпка от резолюцията (унификация на цел с глава на клауза) един процес се заменя със система от процеси (в частност с празен процес, при което изпълнението на програмата завършва). Общите променливи в целите играят ролята на комуникационни канали между тях. Тази интерпретация е в основата на паралелните и обектно-ориентираните подходи при реализацията на Пролог.

1.2. Подходи към паралелна реализация на Пролог

Повечето паралелни схеми на логическото програмиране се основават на идеята за едновременно (в някакъв смисъл) изпълнение на целите в конюнкцията (И-паралелизъм) или на всички клаузи, кандидати за унификация с целта (ИЛИ-паралелизъм). Реализацията на тези идеи обаче е свързана с редица трудности, които са основна причина паралелните версии на Пролог да имат повече теоретично, отколкото практическо значение. Една очевидна теоретична трудност например е реализацията на И-паралелизма при наличието на съвместени (shared) променливи в целите, което изисква предаване на параметри между процесите, изпълняващи отделните цели в конюнкцията. ИЛИ-паралелизмът създава по-скоро реализационни трудности поради необходимостта от поддържане едновременно на няколко контекста (свързвания на променливите), в които работи една и съща цел, които трябва да се комбинират с реализацията на И-паралелизма.

За решаване на проблемите на паралелните реализации на Пролог обикновено се следват два пътя: създаване на сложни механизми за автоматично паралелно изпълнение на чисто логически програми или разширяване на изразните средства на езика с цел явно управление на паралелното изпълнение в зависимост от семантиката на програмата. Първият път трудно се постига на практика, тъй като при него е необходим анализ на цялата програма, за да се установят посоките на предаване на параметрите и начинът на синхронизация на целите. Вторият път е

свързан с добавяне на управляващи изпълнението езикови елементи, които имат съвсем неясна семантика в контекста на логическото програмиране.

Най-често се разглеждат следните видове паралелизъм:

1. Ограничен И-паралелизъм. Едновременно се изпълняват няколко независими цели (такива, които нямат съвместени променливи) в конюнкцията.

2. Потоков И-паралелизъм. Едновременно се изпълняват две цели, съдържащи съвместена променлива. Това става чрез стъпково обменяне на стойността на общата променлива между процесите, реализиращи целите.

Наличието на недетерминизъм в логическите програми допуска още две форми на паралелизъм:

3. ИЛИ-паралелизъм. Едновременно се прави опит за удовлетворяване на няколко алтернативни клаузи при изпълнение на дадена цел.

4. Изчерпващ (All-solutions) И-паралелизъм. Едновременно се изпълняват няколко цели в конюнкцията, като всяка цел дава различни решения.

Изброените теоретични проблеми и възможни пътища за решаване на задачата за паралелна реализация на Пролог доведоха до няколко практически реализации, най-известни от които са Concurrent Prolog [64], PARLOG [24] и Guarded Horn Clauses (GHC) [73,74]. Общ преглед на проблемите при паралелната реализация на недетерминизма в логическите програми е направен в [71].

Основни елементи на паралелното програмиране са *процесите, комуникацията* и *синхронизацията между тях*. И трите посочени версии на Пролог използват едно и също съответствие между тези елементи и основните конструкции на езика Пролог.

Процеси. Активирането на всяка цел е еквивалентно на създаването на нов процес. Аргументите на целта са данните, които се обработват от този процес. В този смисъл активирането на една клауза е еквивалентно на заменянето на процес, унифицируем с главата на клаузата, с процесите - цели в тялото й. По време на изпълнение на програмата се извършва редуциране на броя на процесите при унифициране с клауза с празно тяло, докато се стигне до празен процес (ако програмата завършва).

Комуникация. Съвместените променливи в клаузите са каналите за комуникация между процесите. Тъй като логическата променлива *получава еднократно стойност* (single-assignment), тя е еквивалентна на двупосочен комуникационен канал, способен да разпространява само едно съобщение. Съвместените променливи осъществяват предаването на информация между целите в една конюнкция. По този начин една конюнкция от цели дефинира *мрежа от процеси*.

Синхронизация. Най-общо, клаузите в една програма съответстват на правилата, които описват поведението на процесите - създаване, комуникация и синхронизация между тях. Създаването и комуникацията между процесите се дефинира чрез стандартните изразни средства на езика Пролог, докато за описание на синхронизацията между процесите се въвеждат допълнителни изразни средства - т.нар. *управляващ език*. Със средствата на този език се дефинира и схемата на паралелно изпълнение (четирите форми на И и ИЛИ паралелизъм). Синхронизацията между процесите всъщност означава управление на И- и ИЛИ-паралелизма, което става явно чрез управляващия език.

▪ Управление на ИЛИ-паралелизма. Управляващият език и на трите разглеждани паралелни версии на Пролог се базира на формализма, наречен *клаузи на Хорн с предварителен тест* (Guarded Horn Clauses - GHC) и предложен от Ueda [73]. Семантиката на този формализъм се разглежда в [69]. Една клауза с предварителен тест се представя по следния начин:

$$R(T_1,\ldots,T_k) \leftarrow G_1,\ldots,G_m \mid B_1,\ldots,B_n$$

При декларативното разглеждане на GHC операторът "|" (commit operator) се чете като конюнкция и следователно декларативното значение на GHC е същото като на нормалните Хорн клаузи, т.е. $R(T_1,\ldots,T_k)$ е вярно, ако всичките $G_1,\ldots,G_m,B_1,\ldots,B_n$ са верни. Процедурното значение на $B_1,\ldots,B_n$ е като това на тялото на Хорн клаузите, докато конюнкцията $G_1,\ldots,G_k$ се разглежда като предварителен тест (guard) към унифицирането на текущата цел с главата на

клаузата, която установява дали една кандидат-клауза трябва да се изпълни или не (да се активират процесите в тялото ѝ - $B_1,\ldots,B_n$). Обща черта на PARLOG, Concurrent Prolog и GHC е, че конюнкцията $G_1,\ldots,G_k$ не може да свързва променливи. Това е удобно за организацията на т.нар. *ИЛИ-паралелизъм с фиксиран избор* (commited-choice OR parallelism), при който предварителните тестове на всички кандидат-клаузи (клаузи, чиито глави се унифицират с текущата цел) се изпълняват паралелно, докато една от тях успее, след което изпълнението на другите алтернативи се преустановява. Този вид ИЛИ-паралелизъм може да се разглежда като *паралелно търсене на едно решение*, за разлика от пълния ИЛИ-паралелизъм - *паралелно търсене на всички решения*.

▪ Управление на И-паралелизма. Управлението на И-паралелизма се осъществява в рамките на конюнкцията от цели в тялото на клаузите. Една от основните идеи в паралелното програмиране на Пролог е невъзможността едно извършено действие да бъде отменено впоследствие. Например, след като е направен избор на клауза, не е възможно да се избере нова алтернативна клауза, както беше обяснено в предишната точка. Това означава, че свързването на променливите не трябва да се извършва веднага след успешна унификация, а да се отлага дотогава, докато стане напълно сигурно, че избраното свързване няма да противоречи на свързванията на съответната съвместена променлива в другите цели от конюнкцията, т.е. няма да предизвика пропадане на цели. При паралелно изпълнение това правило се реализира чрез временно спиране на изпълнението на даден процес (цел), докато не се изпълни някакво условие върху неговите данни (променливите в целта). Условията за изчакване на процесите се реализират чрез проверка на унификацията на главите и допълнителните тестове на съответните клаузи. По този начин се осъществява синхронизация между мрежата от процеси, представяща една конюнкция. Условията за синхронизация се задават чрез условия върху посоката на свързване на съвместените променливи. Променливите могат да се дефинират като изходни (read-only) или входни (write-only). В различните версии изразните средства за това са различни. Например в PARLOG за всяка клауза се задава явно кои аргументи са входни и кои - изходни. Този подход е

близък до т.нар. декларации на типа (mode declaration) при компилаторите на Пролог. В Concurrent Prolog се дефинират явно само изходните променливи, чрез постфиксния оператор "?", който означава, че променливата не може да се свързва. Свързването ѝ може да се осъществи само чрез съответната ѝ съвместена променлива. Например в конюнкцията f1(X?),f2(X), ако клаузата f1 свързва аргумента си, тя ще бъде временно спряна, докато X се свърже от целта f2. В най-простия случай временното спиране означава отлагане на изпълнението на целите, т.е. промяна на реда на последователното изпълнение. В общия случай обаче този механизъм осигурява синхронизация между паралелно работещите процеси в една конюнкция. В горния пример процесът f1 може да работи едновременно с f2, като единствено когато дойде моментът за унифициране на аргумента му, той трябва да изчака f2, който ще подаде необходимата стойност по еднопосочния канал за комуникация X.

Чрез описания механизъм за дефиниране на типа на променливите една конюнкция от цели (типа на комуникационните канали в мрежата от процеси) могат да се реализират всички описани видове И-паралелизъм.

Първите практически реализации на PARLOG, Concurrent Prolog и GHC са направени на последователни машини със симулация на паралелно изпълнение. При тези експерименти е достигната ефективността на най-добрите последователни реализации на Пролог. Например една от реализациите на Concurrent Prolog на VAX е надминала по бързодействие Quintus Prolog [53] (за който се твърди, че е най-бързият в момента последователен Пролог). Сега се правят опити за реализации на описаните схеми на паралелен Пролог върху паралелни архитектури, но те все още не са получили популярност, тъй като самите паралелни машини са сравнително скъпи, а могат да се използват за решаване на доста ограничен клас задачи. Такъв е японският проект за пето поколение компютри, в които базовият език - версия на GHC, близка до Concurrent Prolog, се реализира на паралелна архитектура.

Описаните подходи за паралелна реализация на Пролог в известен смисъл са традиционни, тъй като са аналогични на методите, използвани във функционалното

програмиране и в релационните езици. Нов подход за паралелно програмиране в областта на логиката от първи ред е предложен от Ф. Жоран [28,29]. Това е всъщност една нова парадигма, в която въпросите за паралелна реализация се разглеждат съвсем естествено, без затрудненията, предизвикани от концептуалните неясноти относно И- и ИЛИ-паралелизма. Поради чисто последователната процедурна семантика на Пролог описаните по-горе опити за паралелна реализация само замъгляват декларативната семантика на езика.

Схемата на Ф. Жоран е формализъм, с който се описват програми в областта на клаузите на Хорн и предикатното смятане от първи ред. Една програма се представя като *мрежа от комуникиращи помежду си процеси*. Процесите представят литералите в програмата. Процесите, представящи контрарните литерали, т.е. тези, които се унифицират в процеса на извода, са свързани с комуникационни канали. Основна особеност на тези канали е, че те са двупосочни и комуникацията по тях се извършва чрез унификация на съответните терми без да е необходима синхронизация. Процесите работят паралелно и асинхронно, унифицирайки термите, които се разпространяват по комуникационните канали, докато се постигне общо съглашение между всички процеси за необходимите за унификацията субституции. Такова съглашение може да се постигне, ако целта е изводима, в противен случай мрежата не дава отговор.

За реализацията на описаната схема се използва езикът FP2 [31], който е език за паралелно функционално програмиране. От една страна, той е спецификационен език, т.е. с него може да се опише архитектурата на една паралелна машина, която в случая представлява мрежа от комуникиращи помежду си процеси. От друга страна, той е изпълним език - работата на една описана с него машина може да се симулира. Основни понятия на езика FP2 са *процеси* и *комуникация*. Процесите са крайни автомати, чиито състояния, входни и изходни функции са *терми*. Функцията на преходите се задава със система от правила за преписване на терми (rewrite rules). Всеки процес има комуникационни портове, чрез които той се свързва с други процеси. Комуникацията между процесите е унификация между термите, подавани на портовете им. Важна особеност на

работата на описаната чрез FP2 паралелна машина е, че всички процеси работят *асинхронно* (независимо един от друг).

При моделирането на една програма от клаузи на Хорн чрез FP2 всеки литерал се представя като процес с едно състояние, а само целевият литерал - с две състояния (второто състояние е необходимо за извеждане на резултата от работата на програмата - свързаните променливи). По този начин за реализация на една логическа програма се използва архитектура с много висока степен на паралелност, тъй като всички процеси работят заедно, дори без да се отличават междинни стъпки (локални състояния) в работата им. Апаратната реализация на такава схема е трудна. Всяка отделна програма може да се реализира на конкретна архитектура (например като се съпостави на всеки процес по един процесор), но по-широките класове програми (дори простите случаи на еднакви програми с различни целеви литерали) изискват архитектури, позволяващи динамична промяна на конфигурацията при наличието на сложни връзки между процесорите. Затова основният път засега е симулирането на описаната схема върху последователни машини. (Последователна реализация на FP2 например съществува за компютри SUN.)

Описаната идея за реализация на логическите програми като мрежа от паралелни машини е прецизирана и развита по-нататък в [30,27]. В работата се доказва строго, че семантиките на една мрежа от паралелни машини и логическата програма, която е описана чрез нея са еквивалентни. За целта вместо FP2 за описание на паралелните машини се използува езикът на *операторната алгебра на агенти*. Въвежда се език, описващ т.нар. *ербранови агенти*. Формално семантиката на този език се идентифицира с множеството от успешно генерирани последователности от терми (success traces) от всички агенти в една мрежа. Дефинирано е съответствие между клаузите на Хорн и езика на ербрановите агенти. При това съответствие семантиката на една програма от клаузи на Хорн, представена чрез езика на ербрановите агенти, съвпада с традиционната й семантика - минималния й ербранов модел.

Резултатът, описан в [27,30], макар и чисто теоретичен е важна стъпка

към паралелните реализации на логическото програмиране, тъй като дава ясна и конкретна интерпретация на логическите програми в една чисто паралелна и разпределена изчислителна среда. При наличието на конкретна реализация на такава среда въпросът за паралелно логическо програмиране ще бъде решен при запазване на логическата семантика на програмите. Трудностите възникват при реализация на среда за паралелно програмиране на базата на езика на ербрановите агенти. Това е задача, която поради своята трудност все още се разглежда извън контекста на паралелните архитектури, а само се моделира на класически последователни машини.

1.3. Резюме

В настоящата глава са описани основните понятия на логическото програмиране и неговия основен представител, езика Пролог. Разгледан е и въпросът за разпределена и паралелна обработка на информацията чрез Пролог. Този въпрос досега винаги се е разглеждал от реализационна гледна точка, почти без да се засяга логическата му основа. Целта на тези изследвания е да се използват паралелни архитектури за реализацията на езика, което би повишило ефективността на програмите. Естествено тези реализации засягат и основните концепции на Пролог, въвеждайки нови управляващи конструкции, които често са доста далеч от декларативната му (логическа) семантика. Тук именно се крие и основното противоречие при създаването на паралелни реализации на Пролог - невъзможността да се запази чистата декларативна семантика на езика при въвеждането на паралелно изпълнение.

В настоящата работа се предлага подход към решаването на задачата за реализация на концепциите на логическото програмиране, чрез разпределена схема на изчисления. Изходна точка на работата е реализацията на език за разпределено изчисление, който в една от своите интерпретации се разглежда като език за логическо програмиране.

ГЛАВА 2. Език на мрежовите клаузи

2.1. Увод

В настоящата глава се описва формализъм за създаване на модели за разпределена обработка на информацията (мрежови модели), който ще наричаме *език на мрежовите клаузи*. Някои от основните идеи на формализма са описани от нас в [39]. В настоящата работа те са развити и задълбочени в две направления: разглеждане на мрежовите клаузи като самостоятелен език и използуването им като механизъм за логически извод. Основните резултати по тези две направления са описани в [38,40]. Възможностите на езика на мрежовите клаузи за реализация на една схема за разсъждения по премълчаване са разгледани в [41], а едно приложение на тази схема е описано в [67].

Основните идейни източници при създаването на езика на мрежовите клаузи са два:

1. *Логическото програмиране*. Езикът на мрежовите клаузи взаимства от логическото програмиране алгоритъма за унификация, който се използува като основна процедура както за обработка на данни, така и за управление. Използуван е също и синтаксиса на термите.

2. *Разпределените схеми за обработка на информацията*. Основната идея, използвана в случая, е общата организация на изчислителния процес - множество от самостоятелни обработващи елементи, които обменят помежду си информация при отсъствие на централизирано управление. Другият аспект на разпределената обработка - паралелността, не се разглежда в езика на мрежовите клаузи. Това ни освобождава от трудностите при симулация на паралелно изчисление на последователна архитектура, като в същото време осигурява потенциална възможност за паралелна реализация. (Паралелната реализация на разпределена схема за обработка е по-скоро реализационен отколкото теоретичен въпрос.) Връзката на езика на мрежовите клаузи с разпределената обработка на информацията (конекционизма) е разгледана по-подробно в раздел 5.4.

2.2. Основни символи

Основните символи на езика са *термите*. За тяхното записване се използува синтаксисът на термите в Пролог. Дефиницията на терм е следната (приемаме за дефинирани понятията "атом", "променлива" и "число" в смисъл на езика Пролог):

<терм> ::= <атом> | <число> | <променлива> | <структура>

<структура> ::= <атом> | <функтор>(<последователност от терми>)

<функтор> ::= <атом>

<последователност от терми> ::= <терм> | <последователност от терми> , <терм>

2.3. Мрежови клаузи

Мрежовата клауза е основна конструкция в езика. Тя се дефинира като *последователност от възли, които могат да съдържат съвместени променливи.* В езика на мрежовите клаузи могат да се дефинират няколко вида възли. Всички възли са същински терми, т.е. структури, които не са атоми. Възлите се различават по имената на функторите им.

Променливите в мрежовите клаузи се наричат *мрежови променливи.* Мрежовите променливи синтактично са еквивалентни на променливите в Пролог като частен вид терми. Тяхната семантична роля в езика е да представят връзките в мрежата. Връзките в мрежата се дефинират неявно, чрез цитиране на променливи във възлите. *Участието на една и съща променлива в термите, представящи различни възли, наричаме връзка между тези възли.* Променливите са локални в рамките на една мрежова клауза. Това означава, че явни връзки могат да се задават само в мрежовата клауза. Връзки между различни мрежови клаузи могат да се осъществяват чрез процедурите, които унифицират терми (в частност съвместяват променливи). По-долу следва формалната дефиниция на мрежова клауза.

```
<мрежова клауза> ::= <възел>:[] | <възел>:<последователност от възли>

<последователност от възли> ::= <възел>:<последователност от възли>

<възел> ::= <свободен възел> | <процедурен възел>

<свободен възел> ::= функтор(<последователност от терми>)

<процедурен възел> ::=
      node(<последователност от променливи>,<число>,<процедура>) |
      default(<променлива>,<терм>,<процедура>) |
      default(<променлива>,<терм>) |
      failed(<променлива>,<терм>,<процедура>) |
      failed(<променлива>,<терм>)

<последователност от променливи> ::=
      <променлива> |
      ~<променлива> |
      <последователност от променливи>, <променлива> |
      <последователност от променливи>, ~<променлива>

<процедура> ::=
      <терм> = <терм> |
      функтор(<последователност от терми>) |
      <външна процедура> |
      <последователност от процедури>

<последователност от процедури> ::=
      <процедура>, <последователност от процедури> |
      <процедура>; <последователност от процедури>

<външна процедура> ::= <цел на Пролог> | <метапроцедура>

<метапроцедура> ::= top(<число>) | gen(<число>) |
                    netmode(<число>) | net(<терм>) |
                    lazy(<тип на променливите с отлагане>) |
                    nolazy
```

С описаните по-горе синтактични конструкции може да се описва

структурата на мрежовата клауза. Аналогично на клаузите на Пролог мрежовите клаузи се записват в базата от данни. Чрез външна процедура може да се използува произволна цел на Пролог, включително и неговите вградени предикати. За записване на мрежовите клаузи в базата от данни се използуват вградените предикати на Пролог за достъп до базата (**consult, assert**).

В следващите подраздели се описва семантиката на отделните типове възли на мрежата. Описват се също и метапроцедурите, които задават глобални режими на работа на мрежовите клаузи (**netmode, lazy** и **nolazy**) и показват или променят структурата на мрежата (**net, top** и **gen**).

2.4. Активиране на мрежовите клаузи

Активирането на мрежовата клауза става при *свързване или съвместяване на мрежови променливи*. Това може да се извършва при изпълнение на *процедурите*, участвуващи в процедурните възли. Тъй като тези процедури се изпълняват при определени условия, дефинирани в самите възли, необходима е начална активация на мрежата, която може да стане чрез унификация на свободен възел в мрежовата клауза, в който има мрежови променливи. За тази цел в езика на мрежовите клаузи се използува конструкцията "въпрос".

<въпрос> ::= ?- функтор(<последователност от терми>).

Въпросът може да успее или да пропадне. Въпросът успява, ако процедурата се изпълни успешно (успее). Една процедура завършва успешно, ако логическата функция от всички участвуващи в нея елементарни процедури има стойност "истина". Последователността от процедури, разделени с "," се интерпретира като конюнкция, а с ";" - като дизюнкция. Различните видове *елементарни процедури* успяват при следните условия:

1. <терм1> = <терм2> успява ако терм1 се унифицира с терм2;
2. функтор(<последователност от терми>) успява ако в мрежата съществува свободен възел, които се унифицира с дадената процедура.
3. <външна процедура> успява ако активирането на съответната цел на Пролог успее. Метапроцедурите винаги успяват.

Дефинициите за успех на процедурите се основават на унификацията. Използваният алгоритъм за унификация е еквивалентен на този в Пролог. Съществува обаче съществена разлика във валидността на резултата от унификацията на мрежовите променливи и на логическите променливи в Пролог. Тази разлика се състои в това, че свързването и съвместяването на мрежовите променливи става глобално в рамките на цялата мрежова клауза. Аналогично е поведението на логическите променливи в Пролог, участвуващи в един терм.

Успехът или неуспехът на въпроса се индицира от *интерпретатора на мрежовите клаузи*, като при успех се съобщават свързванията на променливите, участвуващи във въпроса. По този начин *въпросът реализира интерфейса на мрежовата клауза с външния свят.*

Следва пример на мрежова клауза, състояща се само от свободни възли:

```
a(X):b(Y):c(X+Y).

?- a(1),b(2),c(X).
X=1+2

yes
?- c(a+b+c),a(X),b(Y).
X=a + b
Y=c

yes
?- a(1),b(2).

no
?-
```

Подобна мрежова клауза работи пасивно, т.е. при наличие на външни процедури, които унифицират променливите ѝ, като тя само разпространява резултата от унификацията по съвместените променливи. Това поведение отговаря на един терм в Пролог, за който е осигурен глобален директен достъп до аргументите му от различни цели в програмата. Променливите в този прост вид мрежови клаузи могат да се използуват в Пролог като *глобални логически променливи*. Те имат всички свойства на логическите променливи, с изключение на това, че съществуват само в един контекст (копие), достъпен за всички цели в програмата.

2.5. Активиране на процедури във възлите

Различните условия за активация на процедурите, участвуващи в процедурните възли, се подчиняват на едно принципно правило: те зависят само от унификацията на мрежовите променливи. Това ограничение е продиктувано от факта, че езикът реализира *разпределена схема за изчисление без наличието на централизирано управление*. С други думи управлението в езика на мрежовите клаузи се осъществява от процедурата за унификация, която е основната и единствена процедура за обработка на данни, т.е. *управлението се ръководи от данните*.

Съществуват три вида условия за активация, задавани чрез различните видове процедурни възли. Трите вида активация се определят от трите възможни резултата от унификацията - *свързване на мрежови променливи, съвместяване на мрежови променливи и неуспешна унификация*. Тъй като променливите разпространяват данните, очевидно е тези три случая да се обработват, за да може да се реализира и управлението в мрежата.

2.5.1. Разпространяваща се активация

Този вид активация се определя от свързването на мрежовите променливи.

Условието за активация се дефинира чрез възли от следния тип:

node(<последователност от променливи>,<число>,<процедура>)

и зависи само от броя на свързаните мрежови променливи (успешна унификация с терм, различен от променлива). За да дефинираме по-точно условието за активация ще въведем още две понятия.

<проста променлива> ::= <променлива>

<променлива с отрицание> ::= ~<променлива>

Процедурата се активира, ако *разликата от броя на свързаните прости променливи и броя на свързаните променливи с отрицание е равна на параметъра <число>*. На практика този параметър играе ролята на брояч на свързванията. Свързването на проста променлива намалява стойността му с единица, а свързването на променлива с отрицание го увеличава с единица. Така че този параметър на възела може да се използува за динамична индикация за броя на свързванията на неговите входни променливи. Активирането на процедурата става, когато се изпълни условието <число> = 0.

При дефиницията на разпространяващ активацията възел има само едно ограничение - параметърът <число> трябва да е по-голям от 0. Това е естествено ограничение, което осигурява процедурата да не се активира "по дефиниция", а само при свързването на поне една променлива. По този начин се съблюдава основният принцип при реализацията на управлението в езика - разпределено управление, ръководено от данните.

В термините на разпределено изчисление числото играе ролята на *праг*, определящ количеството данни, необходимо за да се активира процедурата. Променливите във възела служат за канали (връзки), по които се разпространяват входните данни (терми), като простите променливи могат да се разглеждат като *активиращи връзки*, а променливите с отрицание - като *подтискащи връзки*.

Семантиката на двата типа връзки е илюстрирана чрез следния пример:

```
/*-------- Входове и изходи на мрежата -------*/
основен_вход(X): допълнителен_вход(Y): изход(Z):
/*---------- Обработващи възли --------------*/
/* 1 */ node(X,~Y,1,Z='възел 1 е активен'):
/* 2 */ node(X,Y,1,Z='възел 2 е активен').

/* Примери за активиране на мрежата */
?- основен_вход(данни1),изход(X).
X=възел 1 е активен

yes
?- допълнителен_вход(данни2),основен_вход(данни1),изход(X).
X=възел 2 е активен
```

Основната идея на примера е възможността за разклонение между пътищата в мрежата в зависимост от наличието или отсъствието на информация. Например на входа X се подават данни, необходими и за двата обработващи модула (възли 1 и 2), а на Y - допълнителни данни, обработвани само от 2.

Описаната схема за активация се нарича разпространяваща се, тъй като процедурата обикновено свързва други променливи, които от своя страна активират други възли от същия тип и т.н., т.е. реализира се процес на *разпространение на активацията по мрежата*.

2.5.2. Активация по необходимост

Условието на този тип активация е свързано със съвместяването на мрежовите променливи. То се дефинира чрез възлите от следния вид:

default(<променлива>,<терм>,<процедура>) или

default(<променлива>,<терм>)

При опит за съвместяване на **<променлива>** с друга променлива X (от мрежата или от процедура извън нея) се активира процедурата (ако има такава). След това (или безусловно при отсъствие на процедура във възела), X се унифицира с **<терм>**. При активиране на възела *<променлива> не се свързва с <терм>*, освен ако процедурата не извърши това. Тук **<терм>** наричаме *стойност по премълчаване* (default value) на променливата, цитирана във възела.

Описаното условие за активация е в известен смисъл обратно на условието при разпространяваща се активация. Променливата във възела **default** играе ролята на канал, по който се разпространяват терми при поискване (съвместяване). Възможно и вторият аргумент на възела **<term>** също да е променлива, така че той може да разпространи искането за съвместяване по мрежата.

Важно е да се отбележи, че термът, който се предава чрез мрежовата променлива, не се свързва с нея. Това означава, че една и съща мрежова променлива може да играе едновременно ролята на канал за предаване и за "поискване" на терми. В такъв случай е ясно, че предаваният терм (термът, с който променливата се свързва) има по-голям приоритет от "поисквания", тъй като веднъж свързана, променливата не може след това да бъде съвместена.

За илюстрация на описаните две схеми за активация да разгледаме следния пример:

```
            in1(X): in2(Y): out(Z):
/* 1 */     node(X,Y,2,Z is X+Y):
/* 2 */     default(Z,S,S is X+Y):
/* 3 */     default(X,X1,(write('X=? '),read(X1))):
/* 4 */     default(Y,Y1,(write('Y=? '),read(Y1))).
```

Горната мрежова клауза описва един суматор (Z=X+Y), който работи едновременно в два режима. Първият е като разпространяващ активацията възел (при наличие на входовете изчислява сумата - активира се възел 1):

```
?- in1(1),in2(2),out(Z).
Z=3
```

При съвместяване на изхода Z се активира възел 2, който от своя страна активира възлите 3 и 4 в зависимост от това дали са свързани променливите X и Y. В случай, че са свободни, стойностите на тези променливи се получават чрез процедурите в съответните възли.

```
?- in1(5),out(Z).
Y=? 10.                    /* числото 10 се въвежда от клавиатурата */
Z=15

?- out(Z).
X=? 2.
Y=? 3.
Z=5
```

Следващият пример илюстрира едновременното активиране в двата режима, което показва, че разпространяващата активация има по-голям приоритет:

```
?- in2(3),out(P),in1(5),out(Q).
X=? 0.
P=3          /* активирани са възли 2 и 4 */
Q=8          /* активиран е възел 1 */
```

От примера се вижда, че е възможно подреждането на променливите по премълчаване в йерархия. Процедурата във възел 2 получава аргументите си също чрез активация по необходимост - възлите 3 и 4. Това е *процедурна йерархия*. Възможна е и друга йерархия, т.нар. *йерархия на стойностите по премълчаване* (default hierarchy). Следващият пример илюстрира този вид йерархия. Тя се получава, когато за стойност на променлива по премълчаване се използува друга мрежова променлива, която от своя страна също има стойност по премълчаване.

```
a(X):b(Y):
default(X,Y,p(Y)):
default(Y,def_Y).

p(X):-write('(p,Y) = '),read((t,X)).
```

В тази програма входно-изходните променливи X и Y могат да получават стойности по премълчаване по три възможни начина:

а) чрез свързване на стойността по премълчаване във възела;

```
?- b(val_Y),a(X).
X=val_Y
```

б) при успешно изпълнение на процедурата във възела (когато не е налице условието в предишната точка);

```
?- a(X).
(p,Y) = t,val_X.  /* Процедурата успява и свързва Y с val_X */
X=val_X
```

в) чрез стойността по премълчаване на стойността по премълчаване (йерархията на стойности по премълчаване). Този случай е в сила, когато не са налице условията от предишните точки.

```
?- a(X).
(p,Y) = fail.       /* Терм, при който процедурата пропада */
X=def_Y             /* Последната стойност се получава безусловно */
```

От примерите, илюстриращи йерархията, се вижда, че приоритетът на начините за предаване на стойностите по премълчаване е от а) към в) в намаляващ ред.

Показаните примери за активация по необходимост илюстрират възможностите на мрежовите клаузи за *извод по премълчаване* и *немонотонен извод*, разгледани подробно в глава 4.

2.5.3. Активация при неуспешна унификация

Този вид активация на процедури се дефинира чрез възел от вида:

failed(<променлива>,<терм>,<процедура>) или

failed(<променлива>,<терм>)

Тази дефиниция се взема под внимание при неуспешна унификация. Това може да се случи при следната ситуация: <променлива> е текущо свързана с терма <терм1>. *Прави се опит за унификация на <променлива> с <терм2>*, който *не е унифицируем* с <терм1>. Следователно унификацията пропада. В такъв случай се използува дефиницията на възела и <терм2> се унифицира с <терм>, след което се активира процедурата (ако има такава). Резултатът от тази последна унификация и от изпълнението на процедурата определя (в конюнкция) окончателния резултат от унификацията на <променлива> с <терм2>.

Активацията при неуспешна унификация може да се използува за обработка на свойството *различие между терми*. Както се вижда от дефиницията, възлите от описания тип се активират при два последователни опита за унификация на една мрежова променлива с два *неунифицируеми* (различни) терма. Ако на мястото на <терм> се използува друга мрежова променлива, тя може да се разглежда като *алтернатива* на <променлива> при опитите за унификация на последната.

Друго приложение на описаната схема за активация е възможността за дефиниране на *семантична унификация*. Това става чрез разширяване на стандартната (синтактична) унификация по определен канал на мрежата (мрежова променлива) със специфични правила. Тези правила се задават чрез процедурата във възела и се прилагат при неуспех на стандартната унификация.

Следват два примера, илюстриращи споменатите по-горе две приложения:

1. Буфер с дължина 3. X може да се свързва най-много с 3 различни терма, които се свързват последователно с X, Y и Z. Запълването на буфера се индицира от разпространяващ активацията възел.

```
a(X):
failed(X,Y):
failed(Y,Z): node(X,Y,Z,3,write('буферът е пълен')).
```

```
?- a(1),a(2),a(3).
буферът е пълен

yes
?- a(1),a(2),a(3),a(4).
буферът е пълен
no
?-
```

2. Семантична унификация. По променливата X могат да се унифицират аритметични изрази с техните стойности.

```
a(X): failed(X,Y,(Y is X;X is Y)).

?- a(2*5+5),a(15).

yes
?- a(10),a(2*5).

yes
?-
```

В раздел 5.3.2 е разгледан по-сложен пример за използуване на активация при неуспешна унификация.

2.6. Пропадане на процедурите

Интересен е въпросът за реакцията на мрежата при пропадане на процедури. Това е случай, показващ несъвместимост на термите, подадени на даден възел. Естествено е информацията за тази локална несъвместимост да бъде разпространена обратно по мрежата към източниците на тези терми за да може евентуално да се генерират други входни данни. За да може тази схема да действува е необходимо да съществуват възможности за алтернативни свързвания

на променливи. Такива възможности се осигуряват при използуване на процедура от типа <функтор>(<последователност от терми>) при наличието на повече от един свободни възела в мрежата, унифицируеми с нея. При активирането на такава процедура тя се унифицира с един от няколкото свободни възли, а при търсенето на алтернативи се използуват и останалите. *Механизмът на търсене е аналогичен на този в Пролог - от горе на долу.*

Процесът на търсене на алтернативни свързвания на променливите се предизвиква от неуспешна унификация. Както и активацията този процес се разпространява по мрежата от променливите. Очевидно променливите са единствените връзки между процедурите - едновременно управляващи и информационни. При *разпространяващата се активация* в най-общия случай променливата реализира съответствие между две множества от процедури.

$$\{P_i; i=1,\ldots,n\} \xleftarrow{V} \{Q_j; j=1,\ldots,m\}, \qquad (1)$$

където P_i са процедури, които свързват променливата V, а Q_j - процедури които се активират при свързването на V. Променливата се свързва при изпълнението само на една P_i, а активира всичките Q_j. Пропадането на процедурите Q_j предизвиква търсене на алтернатива измежду P_i, която да свърже отново V. Съществуват три възможности за реакцията на мрежата при пропадане на процедури в активираните възли. Те се задават като текущ *режим на активация*, валиден за всички мрежови клаузи, чрез метапроцедурата **netmode**. Нейният синтаксис е следният:

netmode(<режим>**)**,

където "режим" е число, което определя режима на активация:

0 - пропадането на което и да Q_j не предизвиква търсене на алтернативни свързвания на V;

1 - поне една процедура Q_j трябва да успее, за да не се предизвиква търсене на алтернативни свързвания на V;

2 - всички процедури Q_j трябва да успеят, за да не се предизвиква търсене на алтернативни свързвания на V (това е режимът по премълчаване);

Чрез използуване на метапроцедурата **netmode** може да се влияе на процеса на разпространение на активацията по мрежата. В този смисъл **netmode** е аналог на предиката за отсичане в Пролог ("!"), с тази разлика, че се отнася за работата на всички разпространяващи активацията възли в мрежата. В случаите, когато е необходимо локално дефиниране на режима на активация за конкретен възел, може да се използува следната конструкция:

node(X_1,...,X_n,M,(netmode(<режим на активация>),<процедура>))

Използуването на търсенето на алтернативи при пропадане на процедури в мрежата ще илюстрираме със следния пример:

```
a(X):a(Y):a(Z):
sorted(X,Y,Z):
node(X,Y,2,X>Y):
node(Y,Z,2,Y>Z).

?- a(5),a(2),a(8),sorted(X,Y,Z).
X=8
Y=5
Z=2

?- netmode(0).     /* в този режим сортировката не работи */

?- a(1),a(2),a(3),sorted(X,Y,Z).
X=1
Y=2
Z=3
```

Мрежовата клауза от примера реализира сортиране на три числа по метода на

"мехурчето". Сортирането става като всяка съседна двойка входни данни (X,Y и Y,Z) се подреждат чрез алтернативни свързвания на променливите, предизвикани при пропадане на процедурите във възлите (тестовете за правилна наредба). Вижда се и влиянието, което оказва режимът на активация на мрежата (примерът работи правилно само при режим 2, който е режимът по премълчаване).

Семантиката на схема (1) може да се разшири при използуването на възли, реагиращи на *неуспешна унификация*. По този начин реакцията на пропадане на процедури може да се обработва без търсене на алтернативи. С други думи получаваме следната *еднопосочна* схема:

$$\{P_i;\ i=1,\ldots,n\} \xrightarrow{V} \{Q_j;R_k;\ j=1,\ldots,m;\ k=1,\ldots,l\} \qquad (2)$$

failed(V,V1,R1):
failed(V1,V2,R2):
...
failed(V1-1,V1,R1).

Q_j са процедури, чието пропадане активира процедурите R_k, при което не се предизвиква търсене на алтернативи измежду P_i. За да работи тази схема процедурите P_i трябва да генерират всички възможни свързвания на V. По такъв начин схема (2) може да се интерпретира като *едновременна обработка на всички алтернативни решения*, докато схема (1) - като *обработка на алтернативи при поискване* (пропадане на някои от процедурите Q_j).

В описаните схеми на обработка на неуспешна унификация в мрежата не разглеждаме активацията по необходимост, тъй като пропадането на активираните процедури при нея не предизвиква пропадане на активиращите процедури. На тази реакция може да се даде следната мотивация: при съвместяване на променливата се търси информация (терм); при неуспех на активираната процедура (подготовката на тази информация) заявката е неудовлетворена, но информационният канал остава свободен (свободна променлива) за евентуално бъдещо подаване на информация (свързване на променливата).

2.7. Динамична промяна на структурата на мрежата

Описаните дотук елементи на езика на мрежовите клаузи реализират разпределена схема за обработка на данни (терми), управлявана изцяло от унификацията. При това мрежовата програма има фиксирана структура, която не се променя по време на работата ѝ. Под структура на мрежата разбираме връзките в мрежата, реализирани чрез съвместените променливи, участвуващи в различни възли. При разпределената обработка обаче често е трудно да се знае предварително (да се препрограмира) точната структура, подходяща за даден тип данни. Следователно необходима е възможност *самите данни да определят структурата на мрежата, която ги обработва*. Разгледан в общия случай този въпрос е труден и се решава в рамките на областта *машинно обучение* [45]. В настоящия раздел се описват възможностите за динамична промяна на структурата на мрежовите клаузи само като елемент на езика. Разглеждането на тези въпроси в по-общия контекст на обучението е направено в глава 6.

В термините на дефинирания език на мрежовите клаузи структурата на мрежата се определя от наличието на съвместени променливи в различните възли. Следователно, необходима е процедура, която да съвместява мрежови променливи. За дефинирането на такава процедура ще използуваме т.нар. *обобщение на терми*. Обобщението се дефинира като *замяна на еднаквите константи (атоми и числа) в един терм с еднакви променливи*. Семантиката на тази операция е следната: ако термите се разглеждат като релации между обекти, то конкретната релация между конкретни обекти се заменя със същата релация между класове обекти. Класовете в случая се асоциират с променливи. Свойството, което се запазва при операцията обобщение, е еднаквост/различие, т.е. основното свойство, което се обработва от унификацията. Ето един пример на обобщение (тук обобщението означаваме както субституция):

E=t(a,b,f(a,b)), g = {a/X,b/Y}, тогава Eg = t(X,Y,f(X,Y))

Вижда се, че обобщението е специален вид замяна на терми с променливи, обратна на субституцията. Най-общо обобщението може да се запише по следния начин:

$$g = \{t_1/X_1, t_2/X_2, \ldots, t_n/X_n\}, \qquad (1)$$

където t_i са терми ($t_i \neq t_j$, $i \neq j$), а X_i - променливи ($X_i \neq X_j$, $i \neq j$). Тази дефиниция е частен случай на дефиницията за *обобщение на терми* (term generalization) [34], където един терм t_1 се определя като *по-общ* от друг терм t_2, ако съществува субституция s, така че $t_1 s = t_2$.

Да разгледаме следното множество от терми:

$$S = \{ t(a,b,f(a,b)),\ t(b,c,f(b,c)),\ t(c,d,f(c,d)),\ldots \},$$

получено чрез замяна на константите в терма t(a,b,f(a,b)). Термите от S са *основни примери* (ground instance) на t(X,Y,f(X,Y)). А термът t(X,Y,f(X,Y)) се получава при прилагането на обобщение към кой да е член на множеството S. Следователно:

$$Sg=\{t(X,Y,f(X,Y))\},$$

т.е. g редуцира множеството S до един елемент. Този пример ни дава основание да разглеждаме обобщението като аналог на *най-общия унификатор* (mgu).

Операцията обобщение може да се дефинира по-общо, като вместо замяна на еднакви константи се прави замяна на унифицируеми подтерми. Например ако g е такъв вид обобщение и E=t(a,f(b),f(b)), то:

$$Eg=t(X,f(Y),f(Y))=E_1;\ E_1g=t(X,Z,Z)=E_2;\ E_2g=t(X,X,X)$$

Вижда се, че в този случай обобщението не дава еднозначен резултат, тъй като не е ясно до кога да се прилага унификацията към подтермите. Ако се приеме

правилото унификацията на подтерми да се прилага докато е възможно, то се стига до прекалено опростяване на структурата на терма, т.е. до загуба на информация, зададена чрез функционалните символи в термите.

За да се запази еднозначността на обобщението в по-общата си дефиниция то може да се разглежда като двуаргументна операция. В този случай тя се дефинира по следния начин: *при успешна унификация на два терма се получава трети чрез замяната на различните променливи, свързани с унифицируеми терми, с еднакви променливи.* Следният пример илюстрира обобщението между два терма:

$$\{t(a,b,p(a),t(X,Y,p(Z))\}g=t(X,Y,p(X))$$

Вторият терм играе ролята на образец, по който се изгражда обобщението. Общият алгоритъм за построяване на обобщението g на терма T по образеца P е следният (V_i и W_j са променливи):

Нека $Ts = P$, $s = \{V_1/t_1, V_2/t_2, \ldots, V_n/t_n\}$, тогава
$g = \{t_1/W_1, t_2/W_2, \ldots, t_n/W_n\}$, където (2)
W_i е еднаква с W_j, ако t_i се унифицира с t_j; $i,j=1,\ldots,n$

В езика на мрежовите клаузи процедурата за обобщение се основава на дефиниция (2). Ролята на P се изпълнява от мрежовата клауза, а T е конюнкция от процедури, която се унифицира със свободни възли от мрежовата клауза. Множеството s в случая се състои от свързвания на мрежови променливи, получени при унификацията на T с P. За определянето му се използуват двете *метапроцедури* - top(<маркер>) и gen(<маркер>). Първата маркира началото на създаване на множеството s, като връща в аргумента си стойността на маркера. Изпълнението на втората процедура с аргумент същия маркер означава извършване на обобщението g с текущо получените свързвания от маркираното с top начало (s). Резултатът от обобщението е модифицираната мрежова клауза, т.е. образецът P се използува за съхраняване на резултата от прилагането на g към T. Това означава, че след извършване на обобщението P=Tg.

На практика процедурата **gen** *съвместява мрежови променливи, които са се свързали с унифицируеми терми при работата на мрежовата програма, а комбинацията с* **top** *се използува за дефиниране на областта на действие на тази операция.* Следният пример илюстрира работата на процедурата за обобщение в езика на мрежовите клаузи. В примера се използува метапроцедурата **net(<терм>)**, която унифицира аргумента си с текущата мрежова клауза и се използува за показване на текущите мрежови клаузи в базата от данни.

```
a(X):[].                              /* Мрежова клауза 1 */
b(Y):node(Y,1,write(Y)).              /* Мрежова клауза 2 */

?- a(1).                              /* X и Y са различни променливи */
                                      /* свързването на X не се отразява на Y */
yes
?- top(X),a(1),b(1),gen(X).           /* Свързване и съвместяване на X и Y */
1

yes
?- a(1).     /* X и Y са една и съща променлива */
1            /* свързването на X се индицира от възела, в който участва Y */

yes
?- net(X).                  /* Мрежовата клауза, получена след обобщението */
X=a(_1) :
  b(_1) :
  node(_1,1,write(_1))
```

В базата от данни са въведени две отделни мрежови клаузи, които нямат съвместени променливи. След свързването на променливите X и Y с един и същи терм (числото 1) и изпълнение на метапроцедурата **gen** те се съвместяват. След това чрез възела a (мрежова клауза 1) може да се свърже променливата в мрежова клауза 2, което се индицира от процедурния възел с отпечатване на свързания терм (числото 1).

2.8. Унификация с отлагане

Основно свойство на логическата променлива, което е запазено и при мрежовата променлива, е, че тя получава стойността си еднократно (single-assignment), т.е. тя е канал, по който може да се предаде успешно само един терм. Следователно всеки разпространяващ активацията възел може да работи също еднократно (тук не става дума за търсене на алтернативни свързвания при пропадане на процедурата). Това е особеност, която прави мрежовите клаузи в известен смисъл "плоски", т.е. всеки възел може да обработва само една комбинация от терми и при наличието на много данни, макар и еднотипни, трябва да се предвидят множество еднакви обработващи възли. Това свойство е типично за конекционизма, където за всяка данна трябва да има и структура (възел), която да я обработва. Този подход не е типичен за символната обработка, където обикновено се използуват унифицирани процедури, които обработват множество еднотипни данни. Пример за такава обработка е клаузата в Пролог, представляваща шаблон за обработка на терми, който може да се прилага многократно.

Тъй като нашият подход към конекционизма е символен, в езика на мрежовите клаузи съществува специален вид мрежова променлива, която реализира идеята за многократно присвояване на стойност (multiple-assignment) при запазване на свойствата ѝ на логическа променлива. Мрежови променливи от този вид наричаме *променливи с отлагане на свързването* (lazy variables), а унификацията, която се реализира на този принцип - *унификация с отлагане* (lazy unification). Идеята за унификация с отлагане е реализирана в езика чрез два типа мрежови променливи с отлагане - тип 1 и 2.

В синтаксиса на езика на мрежовите клаузи променливите с отлагане на свързването се задават в отделни мрежови клаузи. Този вид мрежови клаузи се определя от режима при въвеждането им в базата от данни. За тази цел се използуват метапроцедурите **lazy(N)**, която включва режима на променливи с отлагане на свързването (унификация с отлагане) и **nolazy**, която го изключва. Параметърът **N** може да приема две стойности - 1 и 2, които определят типа на

променливите с отлагане на свързването.

Мрежовите променливи с отлагане на свързването от *първи тип (пълно отлагане на свързването)* притежават три основни свойства, които ги различават от обичайните мрежови променливи:

1. Те не се свързват, а само предават стойността, която са получили на процедурата в съответния разпространяващ активацията възел. По този начин те могат да предават повече от един терм.

2. Унификацията с отлагане винаги успява. Това свойство е непосредствено следствие от свойство 1.

3. Предаването на стойността на променливата (валидността на условието за активация на съответния процедурен възел) става само при успешна унификация на всички участия на променливата (съвместени променливи) в свободни възли в рамките на мрежовата клауза.

Променливите с отлагане от *тип 2 (частично отлагане на свързването)* притежават само последното свойство. Следователно те могат да се свързват и унификацията с тяхно участие може да пропада.

Реализацията на свойство 3 предполага отлагане на свързването на променливата, докато не се унифицират всичките й участия в свободни възли на мрежовата клауза. Подобно свойство на променливите съществува в някои версии на Пролог, например в MU-Prolog [49], където то се реализира чрез *отлагане на целите*. Целта в случая на Пролог е по-адекватна реализация на SLD-резолюцията чрез използуване на гъвкаво правило за избор на цел (computation rule). Ето един пример, илюстриращ необходимостта от отлагане на целите, при който стандартният Пролог дава грешен отговор:

```
a(b).
b(a).

?- not a(X), b(X).

no
?-
```

Верният отговор (X=a) може да се получи при отлагане на свързването на X от първата цел not a(X), докато X се свърже от втората цел b(X). (Същият ефект може да се получи и при размяна на местата на целите във въпроса.)

Първите две свойства на мрежовите променливи с отлагане са взаимствани от т.нар. *изчисление с отлагане* (lazy evaluation), познато и във функционалното програмиране. То е предложено за езика ЛИСП едновременно в [26] и [21]. Изчислението с отлагане дава възможност един функционален израз да се редуцира без да се изчислят предварително стойностите на аргументите му. Реализацията на такъв вид редукция се основава на концепциите за *потоково изчисление* (streams) и *съпрограми* (coroutines). Да разгледаме един класически пример на изчисление с отлагане [24]. Това е функцията **Sameleaves**, която връща стойност "истина", когато двата й аргумента (дървета) имат еднакви имена на листата, сканирани в един и същи ред.

```
Sameleaves(x,y) = Eqlist(Flatten(x),Flatten(y))
```

Функцията **Eqlist** връща "истина", ако аргументите й са еднакви списъци, а функцията **Flatten** връща списък от листата на дървото. При нормалната схема за изчисление първо се изчисляват изцяло аргументите на функцията (дърветата се преобразуват в списъци от **Flatten**), след което получените списъци се сравняват от **Eqlist**. Тази схема е крайно неефективна и не може да работи с безкрайни дървета. При изчисление с отлагане функцията и нейните аргументи-функции се изпълняват постъпково. Стъпката в случая е генериране на нов елемент на списъка. В този случай при първата поява на два различни елемента **Eqlist** ще върне стойност "лъжа" и изчислението ще завърши. При тази схема могат да се обработват и безкрайни дървета.

При потоковото изчисление във функционалното програмиране се използува идеята за *постъпкова комуникация* (incremental communicaton) между съпрограми. Тази идея може лесно да се приложи при реализацията на потоковия И-паралелизъм на паралелните логически програми, тъй като, както е показано в [77] всяка

функционална програма може да се представи като детерминистична логическа програма.

За реализацията на унификация с отлагане в езика на мрежовите клаузи идеята за постъпкова комуникация между съпрограми е развита в нов аспект. Да разгледаме следната мрежова клауза с променливи с пълно отлагане на унификацията:

```
?- lazy(1).
a(X):b(X):node(X,1,(write(X),nl)).
```

На една поредица от цели, унифициращи нейните свободни възли, тя реагира по следния начин:

```
?- a(a(1,2)),b(1),b(2),a(f(X)),b(f(z)),b(a(1,Y)),a(2).
f(z)
a(1,2)
2
X=z
Y=2
```

На тази мрежова програма може да се направи следната *потокова интерпретация*: свободните възли a и b са два порта, на които се подават два потока от терми. Променливата X играе ролята на канал, който унифицира термите от двата потока един с друг. При успешна унификация се удовлетворява условието на възела в мрежовата клауза и се активира процедурата `(write(X),nl)`, която извежда унифицираните терми. От примера се вижда, че термите в потоците a и b не са синхронизирани (не идват по унифицируеми двойки). Това означава, че променливата X осъществява синхронизацията им, т.е. тя реализира *процес на постъпкова комуникация (унификация) между два потока от терми*. Стъпката в случая е обръщение към свободния възел чрез процедура.

Една променлива с отлагане на свързването може да участва в произволен брой свободни възли, т.е. тя може да синхронизира множество от потоци. В този

случай съвместената променлива търси *най-общия унификатор (mgu)* на термите от различните потоци.

Очевидно е, че за реализацията на унификация с отлагане е необходимо термите, които се получават от всеки поток (свободен възел), да се съхраняват за по-нататъшна проверка на тяхната евентуална унифицируемост с термите от другите потоци. За тази цел в режим на унификация с отлагане (и двата типа) към всеки свободен възел се присъединява *локална база от данни*. Това е динамична структура, която няма нищо общо с обичайната база от данни на Пролог, където се съхраняват мрежовите клаузи. Тя съществува само докато мрежовата клауза е активна. Локалната база на свободния възел съхранява всички възможни свързвания на променливите, участвуващи в неговата структура.

Достъпът до локалната база на свободния възел става при обръщение към него чрез *процедура, която има същия функтор и брой аргументи*, като се следват следните правила:

- Когато локалната база е празна, всяка процедура записва своите аргументи в нея.
- Ако аргументите на процедурата се унифицират с някой запис в локалната база не се прави нов запис.
- Ако аргументите на процедурата не се унифицират с никой запис в локалната база, то те се добавят към съществуващите записи.
- Следствие от предишните правила е, че процедура, която се обръща към възел от мрежова клауза с отлагане на унификацията, *винаги успява* (свойство 2 на променливите с отлагане).

Горните правила могат да се илюстрират със следния пример:

```
?- lazy(1).
a(X,Y):[].

?- a(1,2),a(2,1),a(X,Y).          /* Достъп до последния запис (2,1) */
X=2
Y=1
```

```
?-a(1,2),a(2,1),a(1,Y).                    /* Достъп до запис (1,2) */
Y=2
```

Примерът показва една възможност локалната база на свободните възли да се използува като същинска *база от данни* за съхраняване на динамични данни. Такава база предоставя унифициран достъп (запис и четене), основан само на процедурата за унификация. Първият аргумент (X) би могъл да се използува като ключ за достъп, а вторият (Y) - като слот за съхранение на данните. Например:

```
?- a(1,data1),a(2,data2),a(3,data3),          /* Запис на данни */
   a(2,X),a(1,Y).                             /* Достъп до данните */
X=data2
Y=data1
```

При наличие на съвместена променлива в свободните възли тя осъществява унификация между термите, записани в локалните бази на свободните възли, в които участвува. Наличието на унифицируеми терми в отделните локални бази се проверява при всяко добавяне на нов запис в някоя от базите. Успехът на тази проверка може да се индицира *само ако променливата участвува в разпространяващ активацията възел*. В такъв случай, при удовлетворяване на неговото условие, процедурата му се активира. Ако променливата участвува в тази процедура, тя предава терма, който я е свързал, *след което се освобождава*.

Възможно е на една стъпка да се получат няколко унифицируеми записи в локалните бази. В такъв случай процедурата се активира многократно с всички получени свързвания на променливата. Следващият пример илюстрира това:

```
?- lazy(1).
a(X,Y):b(Y):node(X,Y,2,(write(X-Y),nl)).

?- a(a,2),a(b,2),a(c,2),b(2).
a - 2
b - 2
c - 2
```

Трите активирания на процедурата се получават след въвеждане на последната цел b(2), която осигурява условието за унификация на двете съвместени променливи Y във възлите a и b.

Използването на мрежови променливи с пълно отлагане на свързването има едно следствие, което е в противоречие с някои класически особености на езика Пролог. Това е невъзможността да се осъществи достъп до повече от една клаузи с еднакъв функтор и брой аргументи, поради факта, че първата клауза унифицира всички процедурни обръщения. В езика Пролог това е важно свойство, на което се основава алгоритъмът за търсене с възврат (backtracking). Същото свойство може да се използува и в езика на мрежовите клаузи при достъп до свободни възли с обикновени мрежови променливи (пример за това е програмата, описана в 6.3.1) или променливи с частично отлагане на свързването. Всъщност невъзможността за използуване на търсене с възврат при унификация с пълно отлагане не е в противоречие с основните концепции на езика на мрежовите клаузи. Това е така, тъй като *потоковото изчисление (streams) и търсенето с възврат (backtracking)* са *противоположни и взаимно изключващи се схеми за организация на изчислителния процес*. Следователно те не могат да се комбинират в единна концепция и именно поради това са въведени и двата типа мрежови променливи с отлагане.

Достъп до множество свободни възли с еднакъв функтор и брой аргументи може да се постигне при използуване на мрежови променливи с частично отлагане на свързването (тип 2), тъй като те се свързват и следователно могат да предизвикват търсене на алтернативни свързвания при неуспешна унификация (по начина, описан в раздел 2.6).

Семантиката и на двата типа променливи с отлагане е от съществено значение за логическата интерпретация на мрежовите клаузи, която ще бъде разгледана в глава 3. Преди да премина към логиката обаче, да разгледаме и някои други съображения за използуването на променливи с частично отлагане на свързването. Ето един пример:

```
?- lazy(2).
a(X,Y):a(Y,Z):a(Z,X):
node(X,Y,Z,3,write(цикъл-X-Y-Z)):                    /* проверка на цикъл */
a(A,B):                    /* свободни възли за графи с повече от 3 дъги */
...
```

Ако е даден граф, описан с дъгите си - множеството **{a(A,B)}**, горната мрежова клауза може да разпознава наличието на цикъл с дължина 3. Например:

```
?- a(1,2),a(2,4),a(2,3),a(4,1).
цикъл-1-2-4
```

Използуването в случая на променливи с отлагане от тип 2 е съществено поради следните причини:

1. Използуването на обикновени мрежови променливи (без отлагане) би предизвикало грешно задействуване на процедурния възел (поради факта, че свойство 3 на променливите с отлагане не е в сила). Например:

```
?- a(1,2),a(3,1).
цикъл-1-2-3
```

2. Променливи с пълно отлагане на унификацията (тип 1) също не могат да се използуват, тъй като в този случай ще е необходимо преименуване на дъгите на графа с уникални имена (поради необходимостта за достъп до тях). В случай обаче, че графът има повече от 3 дъги, няма да е известно предварително кои от дъгите ще образуват цикъл, за да може да се напише подходяща мрежова клауза, която да го разпознава.

2.9. Резюме

В настоящата глава са описани синтаксисът и семантиката на езика на мрежовите клаузи. Основното предназначение на езика се състои в описанието на схеми за

разпределено изчисление без наличието на централизирано управление на базата на унификацията. Характерните особености на езика на мрежовите клаузи са следните:

- Конструкциите на езика синтактично представляват *терми*.
- Чрез мрежовите клаузи се описват *графови структури (мрежи)*, състоящи се от възли и връзки. Възлите дефинират процедури, които унифицират терми, а връзките играят ролята на канали, по които се разпространяват термите.
- Каналите за връзка между възлите в мрежата са реализирани чрез аналог на логическите променливи в Пролог, които в случая се наричат *мрежови променливи*. Те са основен носител на данните (термите) и управлението в езика.
- Работата на една програма на езика на мрежовите клаузи се състои в *активиране на процедурите*, свързани с възлите на мрежата. Това става чрез локални условия върху резултата от унификацията на променливите, участвуващи във възлите (връзките между възлите). Управлението в мрежата е *децентрализирано* - то се осъществява на локално ниво във възлите на мрежата, като не съществуват управляващи средства в езика извън възлите.
- Програмирането на езика на мрежовите клаузи се състои в задаване на структурата на мрежовите клаузи чрез съвместените променливи във възлите. Възможно е автоматично програмиране (обучение), което се основава на възможността за фиксиране на динамично съвместените променливи по време на работа на мрежата.

ГЛАВА 3. Логически извод чрез мрежови клаузи

Езикът на мрежовите клаузи може да се разглежда като език за обработката на термии. Обработката на термии е в основата на повечето схеми за дедуктивен извод. Например езиците Пролог и Лисп могат да се разглеждат като частни случаи на системи за преписване на термии (term rewriting) [31]. Следователно използуването на езика на мрежовите клаузи за дедуктивен извод е въпрос, заслужаващ изследване. Дедуктивните системи имат ясна формална семантика, която би могла да се използува за изясняване на семантиката на езика на мрежовите клаузи. За тази цел първо ще изследваме връзката на езика на мрежовите клаузи с логическия извод при Пролог.

Процедурната семантика на Пролог се основава на реализацията на SLD-резолюцията. Това е ограничение на резолюцията, която от своята е *правило за логически извод*. Декларативната семантика на Пролог дава интерпретация на клаузите на Хорн, използувайки понятията за "истина", "изводимост" и "опровержение". В този смисъл една програмна клауза (правило на Пролог)

$$A \leftarrow B_1, B_2, \ldots, B_n$$

може да се интерпретира по следния начин:

A е "истина", ако всичките $B_1, B_2, \ldots, B_n$ са "истина". (1)

Логическият извод, основан на SLD-резолюцията, намира истинността на A като изследва последователно истинността на всички B_i. Този процес започва от въпроса в една логическа програма и е известен като *извод, воден от целта* (goal-driven inference или backward chaining). Общата процедурна схема на този процес е: една процедура активира множество процедури. Схемата на работа на мрежовите клаузи при разпространяващата се активация е такава, че множество от свързвания на променливи, които могат да се получат чрез активиране на

множество от отделни процедури, води до активиране на една процедура. Следователно в термините на логически извод процедурната схема на работа на мрежовите клаузи може да се интерпретира обратно на тази на Пролог, т.е. като *извод, воден от данните* (data-driven inference или forward chaining). По-точно и формално този въпрос се разглежда в раздели 3.1 и 3.2.

В раздел 3.3 се разглеждат възможностите на езика на мрежовите клаузи за реализация на резолюция върху множество от формули в неклаузна форма (т.нар. неклаузна резолюция). Тази интерпретация на езика му дава и една по-пълна логическа семантика. Неклаузната резолюция, ограничена върху клаузи на Хорн, се свежда до обичайната клаузна резолюция. В този смисъл езикът на мрежовите клаузи може да се разглежда като разширение на класическите езици, основаващи се на SLD-резолюцията.

Езикът на мрежовите клаузи е реализиран в изчислителната среда на Пролог. Както беше споменато в предишната глава възможностите на Пролог могат се използуват изцяло. Това дава възможност за комбинирано използуване на извод, воден от данните и извод, воден от целта, т.е. смесена стратегия за извод. Общата схема на използуването на такъв извод се описва в раздел 3.4.

Разглежданията в настоящата глава, които интерпретират езика на мрежовите клаузи в областта на логиката от първи ред, се ограничават само до разпространяващата се активация. Това е обусловено от факта, че другите схеми за активация (които също се разглеждат по-нататък в работата) притежават свойства, водещи до *немонотонност* при тяхната логическа интерпретация. Ето защо *под език на мрежовите клаузи в настоящата глава ще разбираме само подмножеството на езика, включващо схемата за разпространяваща се активация.*

3.1. Извод, воден от данните

Интерпретацията на разпространяващата се активация като логически извод, воден от данните, се основава на дефинирането на *съответствие между клаузите на Хорн*

и мрежовите клаузи. В изложението в този раздел използуваме тесния смисъл на понятието *логическа програма* като еквивалент на *програма от клаузи на Хорн.* По-долу описваме правила за преобразуване на отделните класове клаузи на Хорн в мрежови клаузи. По този начин съпоставяме на всяка една логическа програма програма от мрежови клаузи:

- Програмните клаузи

$$a(A_1,A_2,\ldots,A_m) \leftarrow b_1(X_{11},\ldots,X_{1n1}),\ b_2(X_{21},\ldots,X_{2n2}),\ \ldots,\ b_k(X_{k1},\ldots,X_{knk}).$$

се трансформират в мрежови клаузи от вида:

$$node(X_{11},\ldots,X_{kn},t,a(A_1,A_2,\ldots,A_m)):\ b_1(X_{11},\ldots,X_{1n1}):\ b_2(X_{21},\ldots,X_{2n2}):\ \ldots\ b_k(X_{k1},\ldots,X_{knk}).$$

Тук числото t играе ролята на *праг*, определящ необходимия брой свързвания на променливи за активирането на възела. За пълната симулация на логически извод върху клаузи на Хорн прагът трябва да е равен на броя на всички променливи в свободните възли, т.е. $t=k*n$. Случаят $t < k*n$ може да се интерпретира като частичен извод, т.е. $a(A_1,A_2,\ldots,A_m)$ може да е истина, дори ако само част от $b_i(X_{i1},\ldots,X_{ini})$ $(i=1,\ldots,k)$ са истина.

- Целевата клауза

$$\leftarrow a_1(X_{11},\ldots,X_{1n1}),a_2(X_{21},\ldots,X_{2n2}),\ldots,a_k(X_{m1},\ldots,X_{mnm}).$$

се трансформира в мрежова клауза от свободни възли.

$a_1(X_{11},\ldots,X_{1n1})$:

$a_2(X_{21},\ldots,X_{2n2})$:

...

$a_k(X_{m1},\ldots,X_{mnm})$.

- Единичните клаузи в програмата

$b_1(T_{11},\ldots,T_{1n1})$.

$b_2(T_{21},\ldots,T_{2n2})$.

...

$b_k(T_{m1},\ldots,T_{mnm})$.

се представят като въпрос в езика на мрежовите клаузи.

?- $b_1(T_{11},\ldots,T_{1n1}),b_2(T_{21},\ldots,T_{2n2}),\ldots,b_k(T_{m1},\ldots,T_{mnm})$.

При трансформираната по такъв начин програма от клаузи на Хорн доказателството, че целта е "истина", започва с унификацията на подцелите във въпроса на мрежовата програма (единичните клаузи на Хорн) със съответните свободни възли. Чрез свързването на аргументите-променливи на свободните възли се предизвиква разпространение на активацията по мрежата през програмните клаузи до целевата клауза.

За да може да се реализира описаната схема на основата на разпространяващата се активация, е необходимо аргументите на свободните възли, участвуващи в трансформираната програмна клауза (X_{ij}; i=1,...,k; j=1,..n), да са променливи, а целите във въпроса на мрежовата програма да са основни терми (T_{ij}; i=1,...,m; j=1,...,n да не са променливи). По този начин работата на програмата се състои в разпространяване на T_{ij} по мрежата от входовете ѝ b_i до изходите - a_i, като при това се съблюдават условията за тяхната съвместимост (унифицируемост), зададени чрез съвместените променливи в мрежовите клаузи. Това разделяне на променливите и основните терми е характерно за голяма част от логическите програми, но въпреки това ограничава тяхната общност. За да се

избегне това ограничение аргументите-променливи на единичните клаузи могат да се вложат в структури, които да се използуват като аргументи на целите във въпроса на мрежовата клауза. Например **a(X,X)** се заменя с **a(f(X),f(X))**.

Следния пример илюстрира използуването на разпространяващата се активация като схема за логически извод върху клаузи на Хорн. Да разгледаме програмата от клаузи на Хорн:

```
1. p(a,b) <--
2. p(c,b) <--
3. p(X,Z) <-- p(X,Y),p(Y,Z)                                    (1)
4. p(X,Y) <-- p(Y,X)
5. <-- p(a,c)
```

Тази програма се превежда на езика на мрежовите клаузи по следния начин (мрежовите клаузи и клаузите на Хорн са номерирани съответно):

```
1,2. ?- p(a,b),p(c,b).
3.   node(X,Y,Z,3,p(X,Z)):
     p(X,Y):
     p(Y,Z).                                                   (2)
4.   node(X,Y,2,p(X,Y)):
     p(Y,X).
5.   p(a,c):[].
```

Програмата (1) има ясен декларативен смисъл, но въпреки това стандартният Пролог, използуващ SLD-резолюция с фиксирано правило за избор на клауза, не може да намери опровержение за нея. Това се дължи на факта, че главите на клаузите 3 и 4 са от най-общ вид и се унифицират с коя да е подцел в програмата. По този начин независимо от реда, в който са записани, едната от тях винаги ще се игнорира при избора.

Програма (2) се изпълнява успешно от интерпретатора на мрежови клаузи. Тя реализира извод на целта 5 от фактите 1 и 2. За активирането ѝ е необходимо само задаването на въпроса (единичните клаузи) **?- p(a,b),p(c,b).**, който инициира процеса на извод. Успехът на този въпрос означава, че е намерено

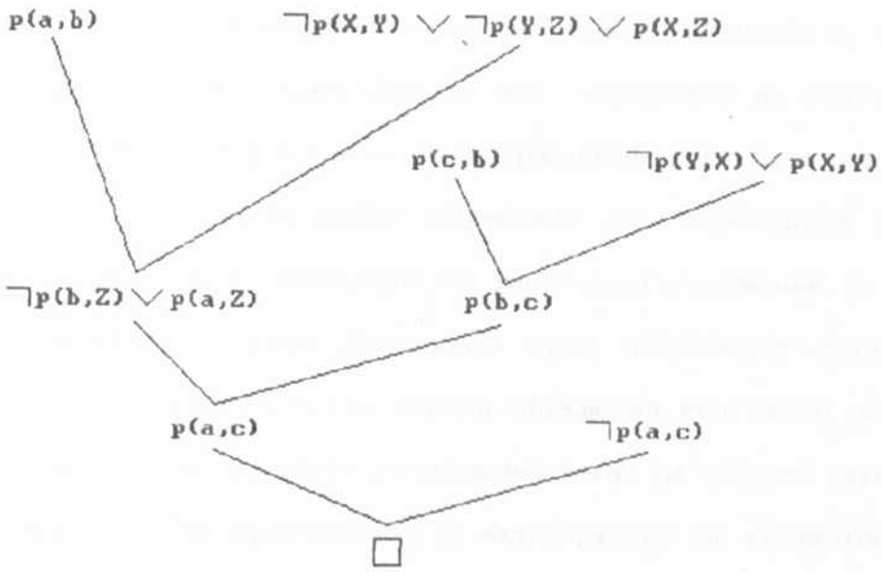
опровержение за програмата (1). Чрез мрежовата програма (2) могат да се доказват и цели, които не са основни терми (каквато е целта във въпроса 5). Например ако модифицираме мрежовата клауза 5 по следния начин:

```
p(X,Y):node(X,Y,2,(write(p(X,Y)),nl)).
```

ще получим възможност да извеждаме множество от твърдения като променяме входните данни или търсим алтернативни решения. Например:

```
?- p(a,b),p(c,b),fail.
p(a,c)
p(b,c)
p(c,b)
p(c,a)
p(b,a)
p(a,b)
no
```

Процесът на намиране на опровержение на програмата (1), воден от механизма на разпространяващата се активация, е показан на фигура 1. Процедурата, илюстрирана с дървото на опровержение от фигурата и изпълнявана на базата на разпространяващата се активация чрез мрежовата програма (2), реализира *стратегия на поддържащото множество* (set of support strategy) [9,70]. Това може да се покаже по следния начин: да означим всички програмни клаузи с S (клаузите 3 и 4 в примера). Тогава единичните клаузи (целите във въпросите на мрежовите клаузи) могат да се интерпретират като поддържащо множество T. Това е така, тъй като множеството от клаузи S-T е удовлетворимо. Последното твърдение е непосредствено следствие от теоремата, че едно неудовлетворимо множество от клаузи трябва да съдържа поне една положителна и поне една отрицателна клауза. (S-T не съдържа положителни клаузи.). Всички изведени при резолвирането клаузи са наследници на клаузи от T, което се явява входно множество за резолюцията.



Фигура 1. Дърво на опровержението, базиращо се на стратегията на поддържащо множество

Тъй като стратегията на поддържащото множество е пълна, то *стратегията за извод чрез разпространяваща се активация е също пълна.*

3.2. Извод при недетерминирани логически програми

Описаният пример (програма (1) от раздел 3.1), който илюстрира логическата интерпретация на мрежовите клаузи, отразява ограничен клас от множеството от логически програми. Това са т.нар. *детерминирани програми.* При тях правилото за избор на клауза (search rule) осигурява еднозначен избор на клауза, чиято глава се унифицира с текущата цел, избрана от правилото за избор на цел (computation rule). В термините на мрежовите програми това означава, че към един свободен възел се обръща само една процедура, която се опитва да унифицира аргументите му. В общия случай на логическа програма, която дава алтернативни решения, съществува многозначност при избора на клауза. Тази многозначност се изразява в съответните мрежови програми като *конкурентност на*

процедурните обръщения към свободните възли, т.е. възможност повече от една процедура да се обръща към един свободен възел. Очевидно е, че обичайните мрежови клаузи, в които променливите имат свойството да получават стойност еднократно, не могат да реализират недетерминирани логически програми.

Въпросът за конкурентните обръщения към свободните възли намира естествено решение чрез използуване на *унификация с отлагане*. В този контекст той може да намери своята реализация чрез *потоковата интерпретация* на мрежовите клаузи (раздел 2.8). Всички обръщения към даден свободен възел образуват поток от терми, които потенциално могат да свържат променливите му. Реалното свързване на променливите и активирането на съответния процедурен възел става при удовлетворяване на условията за унифицируемост на съвместените променливи в различните свободни възли (потоци от терми). Това е процес, точно отразяващ съгласуването на съвместените променливи в конюнкцията от цели в тялото на една програмна клауза. Следователно описаното съответствие между клаузи на Хорн и мрежови клаузи може да се използува и за недетерминирани логически програми, като в случая е необходимо да се използуват мрежови клаузи с отлагане на унификацията. Свойствата на този вид мрежови клаузи дават възможност да се реализират и рекурсивни програми.

По-долу следва пример на недетерминирана логическа програма, реализирана чрез мрежови клаузи.

```
/* Програма от клаузи на Хорн */
1. p(X,Y) <-- a(X,Y),b(Y)
2. a(X,Y) <-- c(X),d(Y)
3. a(1,2) <--
4. a(2,2) <--
5. b(2) <--
6. c(a) <--
7. d(2) <--
8. c(b) <--
9. <-- p(X,Y)
```

```
/* Програма от мрежови клаузи */
?- lazy(1).
node(X,Y,2,p(X,Y)):
a(X,Y):
b(Y).                                        /* 1 */

node(X,Y,2,a(X,Y)):
c(X):
d(Y).                                        /* 2 */

p(X,Y):node(X,Y,2,(write(p(X,Y)),nl)).       /* 9 */

?- a(1,2),a(2,2),b(2),c(a),d(2),c(b).        /* 3,4,5,6,7,8 */
p(1,2)
p(2,2)
p(a,2)
p(b,2)
```

Програмата илюстрира конкурентни обръщения към свободен възел. Възелът **a(X,Y)** се опитва за унификация както от външни процедури (целите **a(1,2)** и **a(2,2)**), така и от процедура в мрежата (процедурата на мрежовата клауза 2, която се активира със следните свързвания: **a(a,3)**, **a(b,2)** и **a(a,2)**). Променливата Y в мрежовата клауза 1 филтрира някои от възможните унификации, проверявайки съвместимостта на литералите **a(X,Y)** и **b(Y)**, които трябва да се унифицират с подходящи данни, за да се активира процедурата **p**. Процедурата **p** от своя страна унифицира свободния възел **p** (изпълняващ ролята на целева клауза на Хорн), който активира възел 11, извеждащ решенията на програмата.

Фактът, че мрежовите променливи с пълно отлагане на унификацията не се свързват дава възможност те да се използуват при рекурсивни програми. Ето един пример в това отношение.

```
?-lazy(1).
X+Y: Z-Y:
node(X,1,call(X)):          /* 1 */
node(Z,1,call(Z)):          /* 2 */
node(Y,1,(write(Y),nl)).    /* 3 */
```

Това е програма, която сканира един символен алгебричен израз, като намира всички подизрази с противоположни знаци. Клауза 1 обработва подизразите с операция събиране, а клауза 2 - тези с изваждане. Възел 3 се активира когато променливата Y получи еднакви подизрази с противоположни знаци. Обработваният израз се подава като цел във въпроса на мрежовата програма. Например:

```
?- a+b-c-d+e-b+d.
b
d

?- a+b+sin(X)-c-sin(a).
sin(a)
X=a
```

Описаната схема за извод, воден от данните, има едно ограничение. Тя работи с логически програми, при които всички отрицателни литерали имат уникални имена. Това се дължи на факта, че се използуват променливи с пълно отлагане на свързването, които не позволяват достъп до повече от един свободен възел с еднакъв функтор и брой аргументи. Това ограничение обаче не е съществено, тъй като може да се премахне с просто синтактично преобразуване на логическата програма по следните правила:

1. Преименуват се всички отрицателни литерали, по такъв начин, че да имат уникални имена (например чрез индексация по реда на появата им).

2. Добавят се допълнителни положителни литерали, които да допълнят двойките от контрарни литерали, в които участвуват преименувани отрицателни литерали.

Горните две правила могат де се илюстрират със следния пример:

```
                                    p(X,Y) <-- a1(X,Z),a2(Z,Y)
p(X,Y) <-- a(X,Z),a(Z,Y)            a1(1,2) <--
a(1,2) <--                  ===>    a2(1,2) <--
a(2,3) <--                          a1(2,3) <--
                                    a2(2,3) <--
```

В разглежданията до тук беше описан начинът, по който всяка логическа програма може да се изпълнява от механизма за разпространяваща се активация на мрежовите клаузи. Процесът на извод в този случай е обратен на този, реализиран в Пролог. Съществено е обаче, че се намира същото множество от решения, т.е. двете схеми за извод са еквивалентни в смисъл на декларативната семантика на логическата програма. Изводът, воден от данните, в езика на мрежовите клаузи може да се разглежда и като решение на една по-обща задача: *намиране на възможните цели (положителни литерали), изводими чрез наличните правила (програмни клаузи) от различни множества от данни (единични положителни клаузи)*. Тази по-обща задача се решава чрез два независими процеса:

1. Локален извод на решения (положителни литерали) от разпространяващите активацията възли, разпространяване на тези решения към други мрежови клаузи, които извеждат нови решения и т.н. Този процес се управлява локално от механизма на разпространяващата се активация и завършва, когато се изведат всички възможни решения при наличните данни.

2. Подаване на нови данни към мрежовата програма и следене на изведените решения, дали удовлетворяват поставените условия на задачата. Важно свойство на тази организация е възможността да се извеждат *частични решения*, което е невъзможно при Пролог.

Съществуват и други схеми и езици, реализиращи извод, воден от данните, които обаче се отнасят за *подмножества* на логическите програми. Например в [5] е описан модел на логически извод, воден от данните, работещ върху семантични мрежи, представени чрез двоични предикати (предикати с два аргумента). Важно е да се отбележи, че схемата за извод, воден от данните, в езика на мрежовите клаузи може да се използува при *логически програми от най-общ вид*. За някои класове програми обаче тя ще бъде неефективна (например при наличието на много факти и малко правила и цели). Естествено това важи и за стратегията, използувана от Пролог, която също не е универсална. Най-общо стратегията на извод, воден от данните, е предназначена за решаване на задачи от

конструктивен тип, т.е. как частите изграждат цялото (такава задача се разглежда в глава 5), докато изводът, воден от целта, е предназначен за задачи от *декомпозиционен тип*, т.е. редуциране на задачата на подзадачи (редуциране на целите на подцели). Тъй като езикът на мрежовите клаузи интегрира в себе си Пролог, и двете стратегии за решаване на задачи могат де се използуват в единна среда за програмиране.

3.3. Логическа семантика на разпространяващата се активация

Изводът, воден от данните, описан в предишните раздели, използува ограничено подмножество от мрежови клаузи. Това е така, тъй като клаузите на Хорн имат не повече от един положителен литерал, т.е. съответните мрежови клаузи имат не повече от един разпространяващ активацията възел. Синтаксисът на езика обаче допуска няколко разпространяващи активацията възли да се срещат в една мрежова клауза. Този случай може да се интерпретира в по-общия контекст на *езика на предикатното смятане от първи ред* (first order language).

Дефиниция 1. *Мрежова клауза* е универсално квантифицирана формула от вида $C_1 \& C_2 \&...\& C_m$, $m \geq 1$, където C_i, $i=1,2,...,m$ са клаузи на Хорн от вида $A \vee \neg B_1 \vee \neg B_2 \vee ... \vee \neg B_n$, $n \geq 0$.

Дефиниция 2. *Програма на езика на мрежовите клаузи* N е конюнкция от мрежови клаузи $N=\{N_1, N_2, ..., N_k\}$. Частен случай на мрежова клауза е единичната клауза A_i. Конюнкцията $A_1 \& A_2 \&...\& A_n$ съответства на въпроса $?- A_1, A_2, ..., A_n.$, където A_i, $i=1,2,...,n$ се наричат *аксиоми*.

Както се вижда от дефиниция 1 основната разлика между мрежови клаузи и клаузи на Хорн е възможността за използуване на съвместени променливи в различни клаузи на Хорн, представени с една мрежова клауза. Въпреки че съществува формална процедура за преобразуване на затворени формули в еквивалентна форма като множество от клаузи на Хорн, възможността за извод директно върху формули от предикатното смятане от първи ред е съществено

предимство, тъй като в този случай се запазва оригиналната семантика на формулите, свързана с предметната област, в която те се използуват. За да илюстрираме това ще разгледаме един пример.

Да предположим, че са дадени две клаузи на Хорн, съответно дефиниращи две геометрични фигури - ромб и успоредник чрез техните страни. Страните им са зададени като структури **edge(X,Y,S,L)**, където X и Y са имената на върховете, които страната свързва, S е ъгловия й коефициент (наклона), а L - дължината й. (Това представяне е описано подробно в глава 5.)

```
ромб(A,B,C,D) <-- edge(A,B,S1,L1),edge(B,C,S2,L1),
                  edge(C,D,S1,L1),edge(D,A,S2,L1).

успоредник(G,H,E,F) <-- edge(E,F,S3,L2),edge(F,G,S4,L3),
                        edge(G,H,S3,L2),edge(H,E,S4,L3).
```

Еквивалентният запис на тези клаузи като формула от предикатното смятане от първи ред е следният:

```
( ромб(A,B,C,D) v
  ~edge(A,B,S1,L1) v ~edge(B,C,S2,L1) v
  ~edge(C,D,S1,L1) v ~edge(D,A,S2,L1)    )
&                                                    (1)
( успоредник(G,H,E,F) v
  ~edge(E,F,S3,L2) v ~edge(F,G,S4,L3) v
  ~edge(G,H,S3,L2) v ~edge(H,E,S4,L3)   )
```

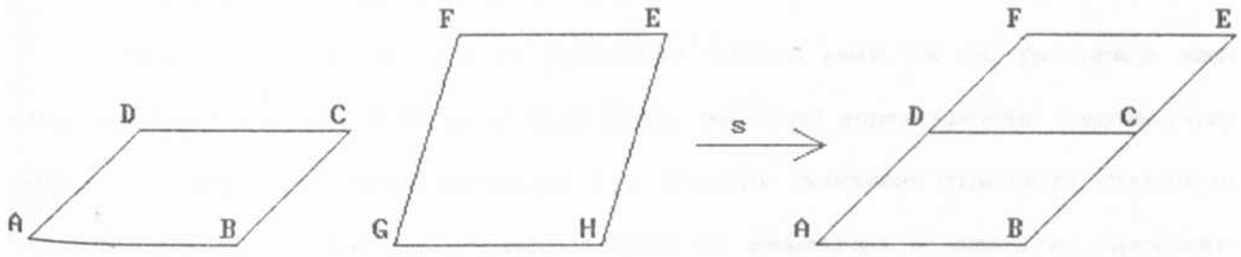
Имайки предвид семантиката на формула (1), към променливите й могат да се приложат следните субституции:

```
s = { G/A, H/B, S3/S1, L2/L1, S4/S2 }
```

Прилагайки s и опростявайки израза, получаваме:

```
( ромб(A,B,C,D) v
  ~edge(A,B,S1,L1) v ~edge(B,C,S2,L1) v
  ~edge(C,D,S1,L1) v ~edge(D,A,S2,L1)   )
&                                                    (2)
( успоредник(A,B,E,F) v
  ~edge(E,F,S1,L1) v ~edge(F,A,S2,L3) v
  ~edge(B,E,S2,L3) )
```

Формула (2) е частен случай на формула (1). За разлика от формула (1) формула (2) не дефинира две отделни геометрични фигури, а само една, която ги обединява. Семантиката на описано преобразуване е илюстрирана на фигура 2.



Фигура 2.

Формула (2) не е вече в клаузна форма и следователно не може да се преобразува обратно в клаузи на Хорн. Според дефиниция 1 обаче формула (2) е мрежова клауза, която се записва в оригиналния синтаксис на езика на мрежовите клаузи по следния начин:

```
?-lazy(2).
edge(A,B,S1,L1):
edge(B,C,S2,L1):
edge(C,D,S1,L1):
edge(D,A,S2,L1):
edge(B,E,S2,L3):
edge(E,F,S1,L1):
edge(F,A,S2,L3):
node(A,B,C,D,4,ромб(A,B,C,D)):
node(A,B,E,F,4,успоредник(A,B,E,F)).
```

Горната мрежова клауза извежда каква фигура (ромб или успоредник) е зададена

чрез даден набор от страни. (В случая се използуват променливи с отлагане на унификацията от тип 2, тъй като всички страни се задават чрез еднотипни структури с еднакви функтори и брой аргументи.) Наборът от страни на фигурата представляват данните, които активират процедурните възли в мрежовата клауза.

Описаният в настоящия раздел пример за разпознаване на геометрични фигури е разгледан подробно в глава 5 като едно от приложенията на езика на мрежовите клаузи. Тук той се използува като илюстрация на възможността да се даде процедурна семантика директно на определен клас формули от езика на предикатното смятане от първи ред, както и да се дефинира логическа семантика на едно подмножество на мрежовите клаузи.

Процедурната семантика на мрежовите клаузи може да се разглежда като ограничение на *неклаузната резолюция* [48], работещо върху мрежови клаузи (клас логически формули според дефиниция 1). По-долу описваме отделните стъпки на неклаузната резолюция във формата, която се реализира в езика на мрежовите клаузи. Да разгледаме следните мрежови клаузи (C_i са клаузи на Хорн, а D_i са аксиоми):

$$(A \vee \tilde{}B_1 \vee \tilde{}B_2 \vee \ldots \vee \tilde{}B_m) \;\&\; C_1 \;\&\; C_2 \;\&\ldots\&\; C_n \qquad (3)$$

$$D_1 \;\&\; D_2 \;\&\ldots\&\; D_k \qquad (4)$$

Да предположим, че литералите $\tilde{}B_1$ и D_1 са контрарни. Тогава, прилагайки правилото на неклаузната резолюция и опростявайки резултатната формула, получаваме:

$$(\; (A \vee \tilde{}B_2 \vee \ldots \vee \tilde{}B_m) \;\&\; C_1 \;\&\; C_2 \;\&\ldots\&\; C_n \;)s, \qquad (5)$$

където **s** е най-общият унификатор (**mgu**) на $\tilde{}B_1$ и D_1. Формулите, получени чрез неклаузна резолюция, се наричат *изведени формули* (derived formulae). Резолвирането на формулите (3) и (4) е първата стъпка от процеса на неклаузна резолюция. Всъщност (4) се представя чрез въпроса на мрежовата програма, който

задействува схемата на разпространяваща се активация.

Да предположим, че на i-тата стъпка на неклаузната резолюция са изведени следните формули:

$$(C'_1 \& C'_2 \& \ldots \& (A_i \vee \tilde{}B_{ik} \vee \ldots \vee \tilde{}B_{im}) \& \ldots \& C'_n)s_1 \qquad (6)$$

$$(A \& C_1 \& \ldots \& C_p)s_2 \qquad (7)$$

Със s_1 и s_2 са означени композициите от последователните субституции, приложени при извеждането на формулите (6) и (7). Да предположим, че $(A)s_2$ и $(\tilde{}B_{ik})s_1$ са контрарни литерали. Тогава, прилагайки правилото за неклаузна резолюция към формулите (6) и (7), получаваме:

$$(C'_1 \& C'_2 \& \ldots \& (A_i \vee \tilde{}B_{ik+1} \vee \ldots \vee \tilde{}B_{im}) \& \ldots \& C'_n)s,$$

където $s=s_1.s_2.s_3$ е резултатната субституция, а s_3 е най-общия унификатор на $(A)s_2$ и $(\tilde{}B_{ik})s_1$. Формули от вида (7) наричаме *предположения*.

Горните разсъждения ни дават основание да дефинираме процеса на неклаузна резолюция, реализиран от схемата на разпространяваща се активация в езика на мрежовите клаузи.

Дефиниция 3. Нека N е програма на езика на мрежовите клаузи, а $\tilde{}G$ е цел. Процесът на опровержение, реализиран чрез неклаузна резолюция в езика на мрежовите клаузи, е последователността от предположения $(A_0)s_0$, $(A_1)s_1$, ..., $(A_n)s_n$, <>, където символът <> означава противоречието (празното предположение), изведено при резолвирането на $(A_n)s_n$ и $\tilde{}G$.

Сега можем да разширим семантиката на извода, воден от данните, в термините на неклаузната резолюция.

Дефиниция 4. *Изводът, воден от данните, в езика на мрежовите клаузи* е ограничение на неклаузната резолюция, при което на всяка стъпка поне една от резолвираните формули е съставена от *аксиоми* (формула 4) или е *предположение* (формула 7).

По този начин процесът на опровергването при дедуктивния извод в езика на мрежовите клаузи се осъществява чрез извод, воден от данните. Изводът, воден от данните, е *коректен* (sound). Може да се докаже също, че този извод е и *пълен* (complete). Използвайки означенията в горните дефиниции, коректността означава, че ако съществува n, така че $(A_n)s_n$ се унифицира с G, то G е логическо следствие от програмата N. Пълнотата на извода, воден от данните, означава, че ако G е логическо следствие от N, то непременно съществува n, така че $(A_n)s_n$ се унифицира с G.

3.4. Смесена стратегия за извод

Важно е да се подчертае, че както Пролог, така и езикът на мрежовите клаузи са преди всичко *езици за програмиране, а не системи за логически извод*. При Пролог това твърдение се обосновава с факта, че при него правилата за избор на цел и клауза са такива, че реализацията на SLD-резолюцията е *непълна* (програма 1 от раздел 3.1 доказва това). При програмирането се използват и някои некоректни в логически смисъл управляващи елементи като отрицанието и символа за отсичане. Всичко това се прави с цел ефективност на програмите, което до голяма степен е постигнато при практическите реализации на езика.

Мрежовите клаузи осигуряват пълнота на логическия извод на базата на стратегията на поддържащото множество върху клаузите на Хорн или върху определен клас от формули от логиката от първи ред (дефиниция 1 от предишния раздел). Резолюцията, която те реализират, обаче е силно неефективна за определен клас от логически програми или формули. Например намирането на опровержение (извода на празната клауза) при тях *не е насочено*, т.е. в процеса на извода се генерират множество "излишни" цели при наличието на "излишни данни". Такива излишни данни например могат да бъдат фактите в една голяма база от данни, което всъщност е нормален случай за логическите програми, особено при използването им като релационна база от данни.

Поради горните съображения считаме, че мрежовите клаузи не са универсална система за логически извод. Те могат да се използуват успешно като такава в някои конкретни случаи. По-пълноценното им използуване обаче като система за логически извод може да стане в комбинация с Пролог. Възможност за това осигурява *пълната съвместимост на данните на двата езика, единната схема за унификация и пропадане на цели, и общата схема за обръщение към процедури.* По този начин във всеки конкретен случай може да се използува най-удобния и ефективен начин на извод.

Различните начини за извод могат да се комбинират в рамките на една програма на базата на следната обща схема:

Например програмата от раздел 3.2 може да се запише по следния начин:

```
/* Програма със смесена стратегия за извод */

p(X,Y):-a(X,Y),b(Y).      /* Клауза на Пролог */

?- nolazy.
a(X,Y):[].                /* Междинни възли за обмен на данни */
a(X,Y):[].
a(X,Y):[].
a(X,Y):[].
b(X):[].

?- lazy(1).
node(X,Y,2,a(X,Y)):       /* Мрежова клауза */
c(X):
d(Y).
```

```
?- a(1,2),a(2,2),b(2),c(a),d(2),c(b),!,          /* Данни */
   p(X,Y),write(p(X,Y)),nl,fail.                 /* Цел на Пролог */
p(1,2)
p(2,2)
p(a,2)
p(b,2)
no
```

Съществена особеност на горната програма е, че междинните възли за обмен на данни между двете стратегии за извод са *без отлагане на унификацията*. Това е така, тъй като е необходимо да се симулира база от данни на Пролог, с която да работят нормалните клаузи. Очевидно това не може да стане чрез възлите с отлагане на унификацията, тъй като обръщенията към тях винаги успяват. От примера се вижда, че по този начин се осигурява възможност за търсене на алтернативни решения чрез възврат. Единствено неудобство на тази схема е необходимостта от предварително задаване на достатъчен брой междинни възли. Разбира се това важи само при недетерминираните логически програми, какъвто е и примерът.

Средствата, които предоставя езикът на мрежовите клаузи, могат да се използуват като допълнение към възможностите на Пролог в чисто процедурен аспект. Например обикновените мрежови променливи дават удобна и ефективна реализация на *глобална логическа променлива*, често необходима при обмен на данни между модули, които не са в йерархията на процедурните обръщения. Унификацията с отлагане разширява възможностите на тази променлива като я превръща в *локална база от данни*, удобна за съхраняване на междинни резултати без използуването на предикатите за достъп до базата (assert/retract). По този начин се решава проблемът с т. нар. *страничнй ефекти в Пролог*, свързани с промяната на базата от данни.

Изброените практически аспекти на мрежовите клаузи дефинират една трета семантика на езика (освен мрежовата и логическата) - използуването му като *допълнение и разширение на процедурните възможности на Пролог*.

3.5. Резюме

В настоящата глава са описани възможностите за използуване на езика на мрежовите клаузи за *дедуктивен извод*. Дефинирано е съответствие между мрежовите клаузи и клаузите на Хорн, на базата на което схемата за разпространяваща се активация реализира *логически извод, воден от данните*. Този извод може да се прилага към клаузи на Хорн, както и към по-широк клас формули на предикатното смятане от първи ред. По отношение на използуването му като дедуктивна система езикът на мрежовите клаузи има следните предимства:

- той е разширение на традиционните езици за дедуктивен извод, базиращи се на SLD-резолюцията (например Пролог);

- дава възможност за комбинирано използуване на различни стратегии за извод (извод от данните и извод от целта) в единна програмна среда;

- реализира ограничение на неклаузната резолюция, която е най-общия вид резолюция.

Разглеждането на езика на мрежовите клаузи като дедуктивна система дава възможност да се дефинира строго негова формална семантика, която наред с приложните му аспекти обосновава неговото съществуване.

Схемата за активация по необходимост показва някои интересни свойства, които могат да намерят естествено приложение в контекста на *разсъжденията по премълчаване* (default reasoning), които са част от по-общата схема на *немонотонни разсъждения* (non-monotonic reasoning) [4]. Формално тези проблеми се изследват в областта на некласическите логики [72].

Разсъжденията по премълчаване са доста обща концепция в изкуствения интелект, която едновременно е теоретична схема и реализационен принцип за различните методи за моделиране на човешките разсъждения. Най-често тя се представя по следния начин: *верността на дадено твърдение се приема, когато липсва информация за неговото отрицание*. Основните идеи на концепцията за разсъждения по премълчаване дава Reiter в [56]. Пълно разглеждане на въпроса, както и възможните схеми за реализация на разсъжденията по премълчаване прави пак същият автор в [54].

Съществуват няколко типични случая на използуване на идеята за разсъждения по премълчаване в изкуствения интелект. Един такъв случай е т.нар. *предположение за затворения свят* (Closed World Assumption-CWA). Това предположение е естествено в контекста на релационните бази от данни [55], където то се използува във формата на следното правило: информацията, която не е зададена в явна форма в базата от данни, се приема за невярна. На това правило се основава и реализацията на отрицанието в Пролог, т.нар. *отрицание чрез пропадане* (negation by failure), което определя пропадането на една цел като успех на нейното отрицание.

Разсъжденията по премълчаване най-често се разглеждат в контекста на формалните системи за извод на твърдения (логики), където се използуват във формата на *правило за извод по премълчаване* (default rule). Най-общата форма на това правило е следната:

$$\frac{\vdash\!\!\!/\ P}{Q} \qquad (1)$$

P и Q са твърдения (предикати), а правилото гласи, че *при невъзможност да се изведе P, се приема за вярно Q*. В този синтаксис предположението за затворения свят (CWA) се записва по следния начин:

$$\frac{\nvdash P}{\neg P}$$

Добавянето на правило от вида (1) обикновено прави всяка една непротиворечива система *противоречива*. Друга особеност е, че реда на прилагане на правилата определя кои твърдения са верни. Например да разгледаме теорията:

$$\frac{\nvdash A}{B} \quad \text{и} \quad \frac{\nvdash B}{A}$$

Тази теория се състои само от едно твърдение. Но то е или A или B в зависимост от това, кое от правилата се прилага първо. Нарушава се и друго важно свойство на логиките - т.нар. *разширяемост* (extension property). При предикатното смятане това свойство гласи, че ако една формула е изводима от дадено множество от предположения P, то тя е изводима и от всяко друго множество от предположения P_1, така че $P < P_1$.

Поради изброените по-горе и редица други особености, чисто формалното дефиниране на логики, включващи някаква форма на правилото (1), води до големи трудности. Въпреки това обаче разсъжденията по премълчаване се използват успешно в много практически области на ИИ, предимно в своите конкретни форми, като например споменато по-горе правило CWA. Разглежданията тук също се основават на един практически вариант на разсъжденията по премълчаване, който се оказва удобна форма за интерпретация на някои от възможностите на езика на мрежовите клаузи.

Основа на интерпретацията на активацията по премълчаване в контекста на разсъжденията по премълчаване е т.нар. *присвояване на стойности по премълчаване* (default assignment to variables), използвано в схемите за представяне на знания и фреймовите езици [6,57]. Присвояването на стойности по

премълчаване може да се дефинира във формата на правило (1) по следния начин:

$$\frac{\nvdash \exists y P(x1,\ldots,xn,y)}{P(x1,\ldots,xn,\langle\text{стойност по премълчаване на } y\rangle)} \qquad (2)$$

Това правило се прилага в процеса на извод, когато опитът да се намери стойност за променливата y, удовлетворяваща предиката P, не е успешен. То дефинира, че в такива случаи се приема стойността по премълчаване на y.

Възелът от типа **default** в мрежовите клаузи може да се използува за дефинирането на подобно правило, което ще се прилага в процеса на извод, управляван от механизма на разпространяваща се активация. Това е извод, воден от данните, при който на всяка стъпка се извършва унификация на процедура със свободен възел в мрежата. Процедурата има формата на предиката P в правило (2), като аргументите ѝ са мрежови променливи. При успешна унификация някои от променливите могат да се свържат, а други - да се съвместят. Термите, свързващи дадена променлива, са предадени от предишни свързвания (предишни стъпки на извода). В тази схема съвместяването на променливата може да се разглежда като *неуспешен опит да се изведе нейната стойност*. Тази интерпретация отразява още по-точно механизма за извод при мрежовите клаузи с отлагане на унификацията. Тъй като при тях унификацията винаги успява, единствената индикация за успех или неуспех на дадена стъпка от извода е свързването на променливите. Ето един пример, илюстриращ това:

```
?- lazy(1).
p(X,Y):[].                      /* Текуща стъпка на извода */

node(X,Y,Z,3,p(X,Z)):           /* Мрежова клауза, осъществяваща извода */
a(X,Y): b(Y,Z).

?- a(1,2),b(2,3),p(X,Y).        /* Успешен извод. X и Y се свързват */
X=1
Y=3.
```

```
?- a(1,2),b(3,3),p(X,Y).        /* Неуспешен извод */
X=_1                            /* X и Y остават свободни */
Y=_2
```

Активацията по необходимост се задействува именно в случая, когато мрежовите променливи останат свободни. В този случай чрез въэли от тип **default** могат да се дефинират стойности по премълчаване на променливите. В горната програма това може да стане като се добавят два въэела към мрежовата клауза, дефинираща текущата стъпка на извода. Например:

```
p(X,Y): default(X,dX): default(Y,dY).
```

След тази модификация, при неуспешен извод променливите ще получават своите стойности по премълчаване (**dX** и **dY**).

```
?- a(1,2),b(3,3),p(X,Y). /* Неуспешен извод */
X=dX
Y=dY
```

Горните разглеждания, илюстрирани с примера, водят естествено до следната модификация на правило (2), с цел прилагането му в езика на мрежовите клаузи:

$$\frac{\not\vdash\ (\ p(X1,\ldots,Xn,Y)\ \&\ \mathbf{nonvar}(Y)\ )}{p(X1,\ldots,Xn,\langle\text{стойност по премълчаване на } Y\rangle)} \qquad (3)$$

Семантиката на правилото може да се резюмира така: *при неуспешен опит за свързване на една мрежова променлива, се приема нейната стойност по премълчаване*. На езика на мрежовите клаузи това се записва по следния начин:

p(X1,...,Xn,Y): default(Y,⟨стойност по премълчаване на Y⟩). (4)

Използувайки особеностите на активацията по необходимост (раздел 2.6.2),

предложената схема за извод на стойност по премълчаване получава редица нови качества в сравнение с оригиналната схема за присвояване на стойности по премълчаване (правило 2):

1. Стойността по премълчаване може да е друга мрежова променлива. Това е начин за дефиниране на връзка (канал) в мрежата по премълчаване. Тук смисълът на правилото за извод по премълчаване е: *в случай, че данните не могат да се получат по даден канал, да се използува друг канал.*

2. Директно следствие от 1 е, че правилата за извод по премълчаване могат да работят в *йерархия*. По този начин може да се реализира наследяване на свойства във фреймови структури за представяне на знания или в семантични мрежи.

3. Стойността по премълчаване не е свързана с предиката, както е в оригиналното правило (2), а със самата мрежова променлива. Всъщност мрежовата променлива е аргумент на предиката. Следователно правилото (4) може да се задействува при частичен успех (неуспех) при извода на предиката. (В термините на извод, воден от данните, частичен успех (неуспех) при извода на даден предикат означава, че само част от аргументите му са свързани.)

Имайки предвид пълната дефиниция на възела за активация по необходимост (с включена в него процедура), може да се въведе следната по-разширена версия на правило за извод по премълчаване:

```
p(X1,...,Xn,Y): default(Y,<стойност по премълчаване на Y>,
                         <процедура по премълчаване за Y>).      (5)
```

Процедурата в правило (5) придава допълнителна гъвкавост на извода по премълчаване. Това е възможността от инициране на нов извод, воден от данните (чрез активиране на процедурата), за намиране на стойността по премълчаване.

Важна особеност на извода по премълчаване е неговата *немонотонност*. Това е свойството при добавяне на нови факти някои преди това изведени твърдения да се окажат неверни. Немонотонността при мрежовите клаузи се определя от

качеството на мрежовите променливи да получават стойност, която припокрива стойността по премълчаване. Това поведение се постига чрез механизма на получаване на стойност по премълчаване - *мрежовата променлива не се свързва със стойността си по премълчаване, а само я разпространява по мрежата при унификацията си с други променливи.* По този начин следващо свързване на променливата ще елиминира стойността по премълчаване, като предаде по мрежата новата стойност. Следващият пример илюстрира немонотонното поведение на извода по премълчаване:

```
a(X):b(Y):c(X,Y):
default(X,Y):
default(Y,default_value).

?- a(Default),c(X,Y).                                    /* 1 */

Default=default_value
X=default_value
Y=default_value

yes

?- a(Default),b(fact1),a(fact2),c(X,Y).                  /* 2 */
Default=default_value
X=fact2
Y=fact1
yes

?- a(Default1),b(fact),a(Default2).                      /* 3 */
Default1=default_value
Default2=fact
```

Въпрос 1 показва директно получаване на стойността по премълчаване **default_value** от променливите X (два пъти - чрез възел a и c) и Y (само чрез възел c). Вторият въпрос илюстрира немонотонността на получаването на стойност по премълчаване. Променливата **Default** получава стойността по премълчаване

(чрез променливата X във възела **a**), след което чрез възлите **a** и **b** се задават стойности на мрежовите променливи. Последната цел унифицира тези стойности, като стойността по премълчаване на **X** (**default_value**) се заменя с новополучената стойност **fact2**. Въпрос 3 показва възможността за получаване на стойност по премълчаване чрез друга стойност по премълчаване чрез йерархия.

Активацията по необходимост може да се използува при реализацията на механизма за наследяване в семантични мрежи. Тук разглеждаме кратък пример от областта на обработката на естествен език, илюстриращ един начин за *интерпретация на сложни съчинителни изречения* с цел намиране на връзка между частите на отделните прости изречения (constituent coordination) [32]. Следната програма осъществява анализ на сложно изречение, съставено от две прости.

```
/*------------- Първо просто изречение --------------*/
o(O1):
v(V1):
p(P1):
delimiter(D1):
node(O1,V1,D1,3,case(V1,O1,P1)):
/*------------- Второ просто изречение --------------*/
o(O2):
v(V2):
p(P2):
delimiter(D):
node(D,1,case(V2,O2,P2)):
/*------- Наследяване на части на изречението -------*/
default(O2,O1):
default(V2,V1):
default(P2,P1):
default(P1,нещо).
/*----------------- Изходи на мрежата --------------*/
case(X,Y,Z):
node(X,Y,Z,3,(write((действие-X,агент-Y,обект-Z)),nl)).
case(X,Y,Z):
node(X,Y,Z,3,(write((действие-X,агент-Y,обект-Z)),nl)).
```

```
/*------ Интерактивно въвеждане на изреченията -------*/
z:-repeat,write('>> '),readbuf,read(X),proc(X),nl,X=край.
proc((X,Y)):-callw(X),!,proc(Y).
proc(X):-callw(X),delimiter(.),!.
/*---------------------- Речник ----------------------*/
callw(иван):-o(иван).
callw(петър):-o(петър).
callw(петър):-o(петър).
callw(яде):-v(яде).
callw(пие):-v(пие).
callw(пише):-v(пише).
callw(пишат):-v(пишат).
callw(вино):-p(вино).
callw(ябълка):-p(ябълка).
callw(круша):-p(круша).
callw(X):-delimiter(X).
callw(_):-!.
```

Входове на мрежата са възлите **o, v, p** и **delimiter.** На тях се подават съответните части на простото изречение. В случая разглеждаме съставни изречения, съставени от две прости, така че всеки един от входните възли е дублиран. Тези входове са свързани със съответния възел на простото изречение. При въвеждане на две пълни прости изречения всички двойки входове **o,v** и **p** се попълват със стойности, вследствие на което се задействуват изходите на мрежата. Например:

```
>> иван,яде,ябълка,а,петър,пие,вино.
действие - яде, агент - иван, обект - ябълка
действие - пие, агент - петър, обект - вино
```

При непълно второ просто изречение, поради липсващите части на изречението входните променливи **O2, V2** или **P2** остават свободни. Съответните стойности за тях се вземат от първото изречение (**O1, V1** или **P1**), чрез първите три възела **default** в мрежата. Те се активират при активирането на целта **case** във възела **node** за второто просто изречение (в този момент се прави опит за съвместяване

на някои от мрежовите променливи O2, V2 или P2). Този възел се активира само по една входна променлива (D), свързана при наличие на съюз, подаден на входа delimiter. Ето два примера, илюстриращи този случай:

```
>> иван,яде,ябълка,и,круша.
действие - яде, агент - иван, обект - ябълка
действие - яде, агент - иван, обект - круша

>> иван,яде,ябълка,а,петър,круша.
действие - яде, агент - иван, обект - ябълка
действие - яде, агент - петър, обект - круша
```

Мрежата може да се активира и при непълно първо просто изречение, например:

```
>> иван,пише.
действие - пише, агент - иван, обект - нещо

>> иван,и,петър,пишат.
действие - пишат, агент - иван, обект - нещо
действие - пишат, агент - петър, обект - нещо
```

За реализация на този случай се използува четвъртият възел **default**, който дефинира стойност по премълчаване на обекта в изречението ("нещо"). Както се вижда от втория пример, тази стойност може да се разпространи и до второто изречение. Това е илюстрация за йерархично използуване на активацията по необходимост.

Схемата за извод по премълчаване, описаната в настоящата глава се използува за целите на анализ на естествен език в [67].

В настоящата глава бяха описани възможностите на езика на мрежовите клаузи за реализация на *извод по премълчаване* (default reasoning). Предложената схема попада в общата схема на *присвояване на стойности по премълчаване на променливи* (default assignment to variables), като в същото време показва някои значителни предимства, най-важното от които е възможността за йерархична организация на извода по премълчаване.

ГЛАВА 5. Представяне и обработка на многостени

Едно от естествените приложения на Пролог е структурното представяне на обекти. Под структурно представяне тук разбираме йерархично представяне на обекти чрез под-обекти (части) и отношения между тях. Използването на Пролог за тези цели е естествено, тъй като структурите от данни на Пролог (термите) са йерархични обекти и езикът предоставя един мощен и универсален механизъм за тяхната обработка - унификацията. Пример в това отношение е геометричното моделиране с Пролог [25]. От друга страна един естествен формализъм за структурното представяне на обекти е графовият или мрежов модел. Следователно поне синтактически езикът на мрежовите клаузи предоставя възможности в тази насока.

В настоящата глава се илюстрират възможностите на езика на мрежовите клаузи за представяне и обработка на визуални обекти. Като обект за моделиране е избран един клас от визуални обекти - многостените. Многостените са графични обекти, често използвани в геометричното моделиране и обработката на изображения. Многостени могат да се използват и за моделиране на обекти с криволинейни части чрез апроксимация. Те са удобни и в областта на разпознаването, тъй като съществуват добре развити методи за извличане на информация за праволинейните части на изображенията [18]. В областта на САПР класът на многостените се използва при 3D-интерпретация на ръчно въведени скици.

В настоящата глава най-напред е описан един начин за представяне и обработка на многостени чрез Пролог [37], който играе роля на концептуална основа за моделирането им чрез мрежови клаузи.

5.1. Списъчно представяне на многостени

Многостенът може да се разглежда като множество от върхове и ръбове, структурно описано като граф. Например един четиристъпалник може да се представи

чрез следния списък:

```
[edge(1,2),edge(2,3),edge(3,4),edge(4,1)],
```

където структурата edge(M,N) представя дъга в графа, свързваща възлите M и N. Чрез горния списък се представя един много широк клас от обекти. За да се получи по-тесен подклас (например успоредник, ромб и др.), трябва да се използува повече геометрична информация. Това означава, че трябва да се въведат някои атрибути (ограничения) към всеки ръб. За тази цел може да се използува следната структура за представяне на ръбовете:

```
edge(V1,V2,Slope,Length),
```

където V1 и V2 са имената на върховете, свързани с ръб, който има ъглов коефициент Slope и дължина Length. По този начин получаваме *атрибутен граф*. Използувайки това разширено представяне, един конкретен успоредник се представя със списъка:

```
[edge(1,2,0,20),edge(2,3,30,50),edge(3,4,0,20),edge(4,1,30,50)]
```

В това представяне ръбовете на фигурата са разделени в две групи, в зависимост от техните наклони (ъглови коефициенти) и дължини. Тези групи се дефинират неявно чрез атрибутите на ръбовете, имащи еднакви стойности. Една много важна особеност на това представяне е възможността да се дефинира клас от фигури, като вместо конкретните имена на върхове и стойности на атрибутите се използуват съвместени променливи. По този начин горният списък, представящ един конкретен успоредник, може да се трансформира по следния начин:

```
[edge(A,B,S1,L1),edge(B,C,S2,L2),edge(C,D,S1,L1),edge(D,E,S2,L2)]
```

В този списък еднаквите имена и стойности са заменени с еднакви променливи. Използуването на променливи като атрибути на ръбовете позволява представянето на класа да бъде независимо от конкретните геометрични свойства на

успоредника, като размер, ориентация в равнината и др. Единственото важно за представянето свойство е равенството или неравенството на атрибутите, проверката на което се осигурява от вградената в Пролог процедура за унификация. Конкретните стойности на атрибутите могат да бъдат от произволен тип и измерителни единици. Тъй като успоредността и еднаквата дължина са свойства, които се запазват при паралелно проектиране, описаното представяне може да се използува за разпознаване и на 3D-обекти, зададени с проекциите си.

Нашата цел по-нататък е, използувайки описаната схема за представяне на многостени, да намерим процедура, проверяваща дали даден обект принадлежи към определен клас. В терминологията на графовите модели такава процедура е откриването на изоморфизъм между графи. В разглеждания случай може да се използува следната идея: един граф е подграф на даден граф, ако множеството от дъгите на първия граф е подмножество на множеството от дъгите на втория граф. По време на изпълнение на процедурата за проверка на свойството "подмножество" променливите, представящи възлите и атрибутите на класа, се свързват със съответните конкретни стойности, представящи възлите и атрибутите на конкретния обект. След това се проверява съответствието на тези свързвания с ограниченията, наложени от съвместените променливи в списъка, представящ класа, и при несъответствие се предизвиква възврат и търсене на нови свързвания. Целия този процес е скрит в рекурсията, използувана при дефиницията на показания по-долу предикат **match**, който е модификация на предиката **subset** (подмножество), описан в [12].

```
match([A|X],Y):-member(A,Y),match(X,Y).
match([],_):-!.

member(edge(X,Y,S,L),[edge(X,Y,S,L)|_]).
member(edge(X,Y,S,L),[edge(Y,X,S,L)|_]).
member(X,[_|T]):-member(X,T).
```

Задачата за изоморфизъм на графи е от експоненциална сложност (NP-пълна), както е показано в [22,75]. В някои случаи обаче може да се намери подходящо

представяне, така че алгоритъмът за изоморфизъм на графи да е приложим на практика. Такъв практически аспект на проблема се разглежда в [18]. Целта е да се минимизира броят на стъпките на възврат, които се изискват за решаване на задачата при "лоша" наредба на дъгите в графа или "лошо" именуване на върховете. В [37] е разгледана сложността на задачата в контекста на Пролог и е показано как може да се повиши ефективността на алгоритъма чрез добавяне на повече атрибути или йерархия в представянето. Съществува, обаче и един проблем от "втори ред", който се появява в практиката, когато се използуват няколко класа. Общата ефективност в тези случаи зависи много от реда, в който се проверяват класовете за разпознаване, тъй като наличието на изоморфизъм между конкретния обект и всеки един клас се проверява последователно. Едно решение на този проблем е проверката за изоморфизъм да се организира така, че конкретният обект да се сравнява с всички класове едновременно. Подобна схема се реализира при моделирането на многостени чрез мрежови клаузи.

5.2. Представяне на многостени чрез базата от данни на Пролог

Както беше споменато в предишния раздел, задачата за намиране на изоморфизъм на графи се решава чрез използуване на операции с множества. Освен списъците в Пролог, базата от данни също може да се използува за записване на елементите на множествата. Един елемент принадлежи на такова множество, ако неговото активиране като цел успява. По този начин операцията "подмножество" се свежда до изпълнение на конюнкция от цели. Следната програма например проверява дали една фигура принадлежи към класа на ромбовете.

```
/* Факти, представящи конкретна фигура */
edge(1,2,0,100).
edge(2,3,45,100).
edge(3,4,0,100).
edge(4,1,45,100).
```

```
/* Въпрос на Пролог, дефиниращ класа на ромбовете */
?-edge(A,B,S1,L),edge(B,C,S2,L),edge(C,D,S1,L),edge(D,A,S2,L).
```

Горната схема за представяне на многостени е много по-ефективна от списъчното представяне, тъй като при нея се използуват директно два вградени механизма на Пролог - достъпът до базата от данни и търсенето с възврат. При използуването на това представяне обаче е невъзможно да се проверява изоморфизъм на граф и подграф (напр. липсващи ръбове в конкретния обект, получени като невидими части при проектирането на 3D-обект). Това означава броят на целите в конюнкцията да е по-голям от съответните факти в базата от данни, което естествено ще доведе до пропадане на конюнкцията. Този недостатък е съществен и прави неприложим описаният подход в практиката. Въпреки това представянето на многостени чрез базата от данни лежи в основата на модела за представяне на многостени чрез мрежови клаузи, разгледан в следващия раздел.

5.3. Представяне на многостени чрез мрежови клаузи

5.3.1. Използуване на разпространяваща се активация

Особеностите на мрежовите променливи могат да се използуват съществено в графовия модел на многостените. Освен за представяне на топологията и геометричните ограничения, съвместените променливи в мрежовите клаузи могат да се използуват за представяне на йерархията "част-обект" между върховете и обектите, в които те участвуват, така че след успешна унификация на конкретния обект свързаните променливи (върховете на многостена) показват съответния клас, към който те принадлежат. По този начин се избягва последователното сравнение между проверявания конкретен обект и всички съществуващи класове, което е един от основните проблеми на представянето на многостени чрез списъчни структури на Пролог. По-долу следва програмата (мрежова клауза),

която решава задачата за разпознаване на различни видове четириъгълници. Графичното представяне на мрежата, описваща класа четириъгълници, е показано на фигура 3.

```
/* Входове на мрежата */
edge(A,B,S1,L1):
edge(B,C,S2,L1):
edge(C,D,S1,L1):
edge(D,A,S2,L1):
edge(B,E,S2,L2):
edge(E,F,S1,L1):
edge(F,A,S2,L2):
edge(E,G,S3,L3):
edge(G,A,S4,L4):

/* Изход на мрежата */
fig(Fig):

node(A,B,E,G,4,fig(четириъгълник)):
node(A,B,E,F,4,fig(успоредник)):
node(A,B,C,D,4,fig(ромб)).
```

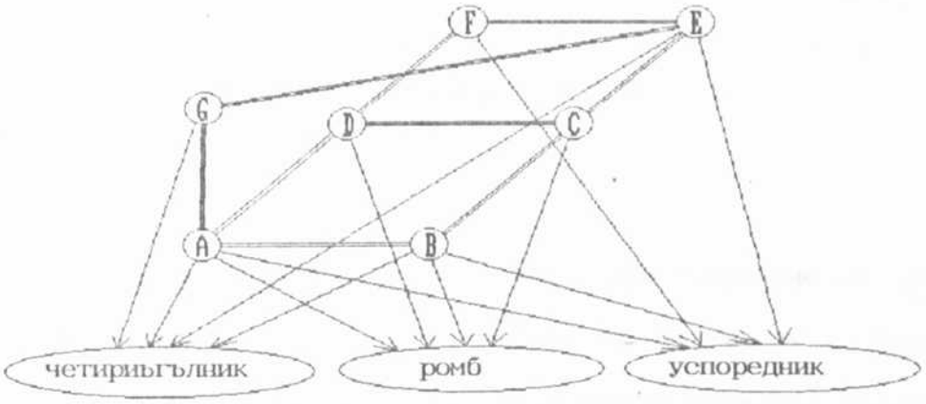
Ето два примера за работата на програмата:

```
?- edge(1,2,0,20),edge(2,3,45,30),edge(3,4,0,20),edge(4,1,45,30),fig(X).
X=успоредник

?- edge(1,2,0,20),edge(2,3,95,20),edge(3,4,0,20),edge(4,1,95,20),fig(X).
X=ромб
```

Описаният пример реализира *еднослоен мрежов модел*, в който не се използуват междинни обработващи модули. Това се дължи на факта, че обработваните обекти са сравнително прости. Основните свойства на описващите ги характеристики са само две - равенство и неравенство. Тези характеристики са общите върхове на ръбовете (представящи топологията на многостена) и класовете, обединяващи атрибути на ръбовете с еднакви стойности (геометричните

ограничения - наклон и дължина на ръбовете). Равенството и неравенството са свойства, които се обработват директно от процедурата за унификация и се разпространяват по връзките на мрежата, без да е необходима каквато и да е допълнителна обработка.



Фигура 3.
Мрежа за представяне на геометрични фигури. Топологичните и връзките "част-обект" са показани с различни линии

Съществуват обаче някои геометрични свойства, които не могат да се изразят в термините на равенство и неравенство. Такова свойство например е перпендикулярността на ръбовете на многостена. За да се провери това свойство са необходими изчисления, които могат да се извършат от междинни възли в мрежата (наречени *скрити възли* в терминологията на конекционизма). На базата на тази идея мрежата от фигура 3 може да се разшири, така че да разпознава и перпендикулярни фигури. Това маже да стане чрез използуване на същите входове и изходи (свободните възли **edge** и **fig**), към които са добавени следните процедурни възли:

```
/* Общ случай на четириъгълник */
node(A,B,E,G,4,fig(четириъгълник)):                    /* 1 */

/* Скрит възел, изчисляващ перпендикулярност */
node(S1,S2,2,p(S1,S2,P)):                              /* 2 */
```

```
/* Неперпендикулярни фигури */
node(A,B,E,F,~P,4,fig(успоредник)):                    /* 3 */
node(A,B,C,D,~P,4,fig(ромб)):                          /* 4 */

/* Перпендикулярни фигури */
node(A,B,E,F,P,5,fig(правоъгълник)):                   /* 5 */
node(A,B,C,D,P,5,fig(квадрат)).                        /* 6 */

/* Процедура за изчисляване на перпендикулярност */
p(X,Y,true):-0 is (X-Y) mod 90,!.
p(_,_,_).
```

Скритият възел в мрежата (възел 2) се активира при свързване на променливите **S1** и **S2** (представящи ъгловите коефициенти на съответните ребове). Ако условието за перпендикулярност е налице, процедурата **p** свързва променливата **P**, като по този начин активира класа на перпендикулярните фигури и подтиска активирането на неперпендикулярните (поради подтискащата връзка **~P**). В противен случай променливата **P** остава свободна. По този начин се активират неперпендикулярните и се подтиска активирането на перпендикулярните фигури. Ето няколко примера за работата на мрежата:

```
?- edge(1,2,0,20),edge(2,3,45,30),edge(3,4,0,20),edge(4,1,45,30),fig(X).
X=успоредник

?- edge(1,2,0,20),edge(2,3,90,20),edge(3,4,0,20),edge(4,1,90,20),fig(квадрат).

yes
?- edge(1,2,0,20),edge(2,3,50,20),edge(3,4,0,20),edge(4,1,50,20),fig(X).
X=ромб

?-edge(a,b,0,20),edge(c,d,45,30),edge(d,e,10,40),edge(e,a,50,60),fig(X).
X=четириъгълник
```

5.3.2. Използуване на активация при неуспешна унификация

Описаните до тук примери работят при успешна унификация на мрежовите променливи. Пропадането на унификацията на дадена мрежова променлива показва несъвместимост на въведената чрез входоцете на мрежата информация. Това от своя страна предизвиква търсене на алтернативни свързвания на входовете на мрежата (променливите във възлите **edge**), т.е. подаване на същата информация на други входове. Използуването на тази схема има един съществен недостатък - трябва да се осигури специална наредба на входовете. Във всички примери с геометрични фигури възлите **edge** бяха задавани по такъв начин, че ръбовете на фигурите с "по-силни" ограничения да се унифицират преди тези на фигурите с "по-слаби" ограничения. (Търсенето на алтернативи става от горе на долу.) Това означава например ръбовете на квадрата и ромба да се въведени в базата от данни преди ръбовете на успоредника. Задаването на специална наредба на входовете на мрежата затруднява особено много създаването на програми, описващи сложни многостени (например в тримерния случай).

Естествено решение на споменатите проблеми за описание на входовете на мрежата е използуването на активация при неуспешна унификация. Следващата програма показва този подход. Тя разпознава успоредници, ромбове, правоъгълници и квадрати.

```
/* Входове на мрежата */
edge(A,B,S,L):
edge(B,C,S,L):
edge(C,D,S,L):
edge(D,A,S,L):

/* Максимум две различни стойности за S */
failed(S,S1,p(S,S1,P)):

/* Максимум две различни стойности за L */
failed(L,L1):
```

```
/* Класове фигури */
node(A,B,C,D,L1,~P,5,fig(успоредник)):
node(A,B,C,D,L1,P,6,fig(правоъгълник)):
node(A,B,C,D,~L1,~P,4,fig(ромб)):
node(A,B,C,D,~L1,P,5,fig(квадрат)).

fig(X):[].

p(X,Y,ok):-0 is (X-Y) mod 90,!.
p(_,_,_).
```

Процедурните възли **failed** се използуват за индикация дали на входовете на мрежата са постъпили различни стойности за наклона (**S**) и дължината (**L**) на ръбовете на фигурата. Ако има такива стойности, то втората от тях се свързва с **S1** (респективно **L1**). Това е индикация за типа на фигурата - с еднакви или с две по две различни страни. Втората характеристика на фигурите - перпендикулярността, се получава от разликата между стойностите на **S** и **S1**, която се изчислява от процедурата **p** и се предава чрез мрежовата променлива **P**.

Описаната програма е по-кратка и значително по-ефективна от съответната ѝ, описана в предишния раздел. Това се дължи на по-малкия брой входове на мрежата и *отсъствието на търсене на алтернативи* в процеса на тяхната унификация. Настоящата програма е пример на мрежа с разпространяваща се активация, без каквато и да е зависимост от реда на записване на входовете или въвеждане на входните данни. Ето защо по отношение на предишните примери тя е "най-близо" до чистите модели за разпределена обработка и "най-далече" от алгоритмите на Пролог за решаване на подобни задачи.

5.4. Език на мрежовите клаузи и конекционизъм

При решаването на задачата за представяне на многостени чрез езика на мрежовите клаузи бяха използувани някои елементи, типични за конекционистките

модели. Описаната програма представлява мрежа, съставена от части на представяните обекти, свързани с ограничения и топологични връзки. Използуват се активиращи и подтискащи връзки, както и скрити възли. Мрежата описва едновременно структурата на моделираните обекти и изчислителната архитектура за тяхното разпознаване. Всички тези особености ни дават основание на твърдим, че езикът на мрежовите клаузи може да се използува за създаване на конекционистки модели. В настоящия раздел разглеждаме по-подробно особеностите на този тип модели и тяхната връзка с конструкциите на езика.

Съществуват различни гледни точки относно конекционистките мрежови модели, главно групирани около два полюса - *конекционизма като когнитивна теория* и *конекционизма като изчислителна архитектура* [20,10,33,47]. Класическите мрежови (графови) модели в ИИ също се разглеждат в различни аспекти, в известна степен отразяващи същите две крайни гледни точки - първият е използуването на мрежовите модели за *представяне на знания*, а вторият - като *архитектура за паралелно и/или разпределено изчисление*. И двата аспекта на мрежовите модели обаче са представители на символните подходи към ИИ, основаващи се на фон-Нойлмановата идея за разделяне на данните и управлението. Всъщност тези два подхода при използуването на мрежовите модели отразяват именно това разделяне. Тук се крие и една от съществените разлики между символните и конекционистките модели - възможността за интегрирано използуване на конекционистките модели едновременно като средство за представяне на знания и изчислителна архитектура за тяхната обработка.

Описаният в предишните раздели език на мрежовите клаузи е опит да се интегрират по непротиворечив начин двата аспекта на използуване на класическите мрежови модели. В този смисъл той може да се разглежда като средство за създаване на конекционистки модели. За да покажем тези възможности на езика на мрежовите клаузи по-долу описваме основните характеристики на конекционистките модели (отпечатани с курсив), както са представени от М. Арбиб в [3], и тяхната връзка с възможностите на езика:

а) *Използуване на мрежа от активни обработващи елементи, като програмите*

се съдържат в структурата на мрежата. Активните обработващи елементи в езика на мрежовите клаузи са процедурните възли. Условията за активация на тези възли изцяло зависят от структурата на мрежата (топологията на връзките). Програмата на езика на мрежовите клаузи задава начина на свързване на процедурните възли, т.е. структурата на мрежата. Следователно програмите на езика на мрежовите клаузи се съдържат във връзките на мрежата.

б) *Обработващите елементи обикновено имат проста структура. Тя зависи от конкретното приложение, но в повечето случаи обработващите елементи са прагови логически устройства от типа на формалния неврон на Мак Калок и Питс.* Разпространяващият активацията възел е прагов елемент. Той обаче е много по-сложен и развит от формалния неврон, тъй като може да извършва не само числови операции (изчисление на тегловни суми), но и сложни символни операции (например логически извод). Това дава възможност за интегрирането на символни и конекционистки модели в единна програмна среда.

в) *Висока степен на паралелизъм, без наличието на централизирано управление*. Езикът на мрежовите клаузи е реализиран в последователна програмна среда. Неговата реализация обаче е такава, че последователната среда, в която работи, минимално влияе на свойствата на мрежовите модели, които се изграждат чрез него. Всъщност схемата на разпространяващата се активация може при определени условия да симулира паралелно изчисление. Разгледан общо, въпросът за симулация на паралелно изчисление в последователна изчислителна среда се свежда до осигуряването на две условия: децентрализирано управление и независимост на изчислителния процес от реда на постъпващите входни данни. Докато първото условие е вътрешно присъщо свойство на езика на мрежовите клаузи, за реализацията на второто условие е необходимо да се наложат следните ограничения:

1. Трябва да се използува само схемата за разпространяваща се активация. Другите схеми допускат немонотонно поведение на изчислителния процес, за което е присъща зависимост от реда на входните данни.

2. Процедурите в процедурните възли не трябва да причиняват "странични

ефекти" (в смисъл на Пролог).

г) *Семантиката на мрежовия модел се кодира чрез отделни възли (локално представяне) или чрез структурата на активните елементи (разпределено представяне).* И двете схеми за представяне на семантиката на мрежовия модел могат да се реализират в езика на мрежовите клаузи. Това е показано чрез програмата за разпознаване на геометрични фигури (раздел 5.3.1). Семантиката на класовете фигури (техните имена) е представена локално чрез разпространяващите активацията възли, а характеристиките на фигурите (топология, дължина, успоредност или перпендикулярност на страните) са разпределено представени чрез връзките в мрежата.

Описаните по-горе особености на езика на мрежовите клаузи, свързани с основните характеристики на конекционизма, показват, че той може да се използува за създаване на конекционистки модели. Възможно е също и някои класически за невронните мрежи обучаващи процедури да се реализират на Пролог и да се използуват чрез процедурните обръщения в разпространяващите активацията възли. Това обаче е далеч от първоначалната цел, поставена при създаването на езика на мрежовите клаузи - интегрирането на символните и конекционистките методи при решаването на задачи на ИИ. Езикът на мрежовите клаузи наследява от конекционизма общата организация на изчислителния процес, а символните методи разширяват семантиката на възлите и връзките на мрежовите модели. Мрежовите променливи (връзките) могат да разпространяват не само числови стойности (както е при невронните мрежи), но и сложни структури от данни (например думи, както е показано в примера за анализ на изречения от глава 4), а процедурите във възлите могат да реализират символни алгоритми за тяхната обработка. Широките функционални възможности на връзките и възлите се дължат на тяхната реализационна основа - унификацията, която е универсална процедура за достъп и обработка на данни. Следователно поради по-сложните и структурирани данни, които се обработват на базовото ниво, мрежовите клаузи могат да моделират реалните обекти и процеси на по-високо ниво на абстракция, отколкото съответните конекционистки мрежи.

Както беше споменато по-горе, езикът на мрежовите клаузи може да се използува едновременно като формализъм за представяне на данни и като изчислителна архитектура за обработката им. Първият аспект е добре илюстриран от програмата за разпознаване на геометрични фигури (раздел 5.3.1). Тази програма описва мрежа, която е едновременно атрибутен граф, представящ класовете фигури, и алгоритъм, по който се решава задачата за разпознаване. Архитектурният аспект на езика всъщност се състои в разпределения характер на изчислителния процес, който той реализира. Пример в това отношение е изводът, воден от данните (раздел 3.1), който реализира една класическа процедура за символна обработка (резолюцията) в разпределена изчислителна среда.

5.5. Резюме

В настоящата глава е описана подробно задачата за представяне и разпознаване на многостени. Предложени са две решения - чрез стандартния Пролог и чрез езика на мрежовите клаузи. Второто решение илюстрира предимствата на езика на мрежовите клаузи при решаването на задачи за структурно представяне на обекти, частен случай на които е разгледаната задача. Програмата на езика на мрежовите клаузи, която решава тази задача, е всъщност една мрежа, която използува типични за конекционизма елементи, като активиращи и подтискащи връзки и скрити възли (hidden units). Описаният пример, както и направеният преглед на основните свойства на конекционистките модели и тяхното съответствие с езика на мрежовите клаузи, доказват възможностите на езика за създаване на модели за разпределена обработка на информацията (конекционистки модели).

В предишните глави беше описан езикът на мрежовите клаузи. Начините за неговото използуване бяха илюстрирани с примери за решаване на задачата за разпознаване на геометрични фигури. Тези примери показват част от предимствата на мрежовите клаузи. Описаните програми са достатъчно прости за да могат да се обяснят лесно, без да се навлиза в технологията на тяхното съставяне. Очевидно е обаче, че решаването на по-общи задачи от областта на обработката на многостени, особено в тримерния случай, при наличието на десетки и стотици входове и възли на мрежата е невъзможно да става чрез използуваното дотук "ръчно" програмиране. Този проблем е напълно естествен в контекста на моделите за разпределена обработка, където е трудно да се говори за програмиране в класическия (последователен) смисъл. Тези трудности се определят от спецификата на представяне и обработка на информацията при тези модели:

- описанието на моделираните обекти е разпределено в мрежата и алгоритмите за обработка се базират на взаимодействието на голям брой отделни възли по начин, неподдаващ се на локално описание;

- описанието на структурата на нерегулярните архитектури изисква почти винаги развит графичен интерфейс;

- в повечето случаи не е ясна точната структура на мрежата и начините на взаимодействие между възлите, когато се решава дадена задача. Това е така, защото често входните данни определят структурата на мрежата. Така е например в случая с представянето на многостени, разглеждан тук.

Всички тези особености налагат наред с дефинирането на конкретния формализъм да се разработят и съответни *методи за обучение*. В контекста на изкуствения интелект машинното обучение разглежда методи за придобиване на нови знания или организиране на съществуващи. Към задачата за обучението съществуват три основни подхода в зависимост от количеството на априорните знания, вградени в обучаващата се система [45]:

- *подход на невронните мрежи (конекционизма)*. Това е подходът на

"самоорганизацията", който предполага наличието на минимални априорни знания на обучаващата се система. Той се базира на идеята, че една "достатъчно богата" празна структура (среда за обучение) чрез едно правило за обучение, приложено "достатъчен" брой пъти, може да придобие желаното поведение независимо от задачата, която се решава. Класически примери на използването на този подход са *перцептрона* [59] и различните методи, базиращи се на намиране на *дискриминантни функции* [50]. От съвременните конекционистки методи за обучение най-разпространени са методът на обратно разпространение на грешката (back propagation learning) [63] и конкурентно обучение (competitive learning) [62].

- *символно изграждане на понятия* (symbolic concept acquisition). Обучаващата се система конструира символни описания на нови понятия на базата на примери и контрапримери. Типично за този подход е за символните описания да се използват логически методи, дърво на решенията, продукционни правила и семантични мрежи. Примери за този подход са програмата ARCH на Уинстон [79] и алгоритъмът ID3 на Куинлан [52].

- *подход на специфичните знания* (domain speicific learning), при който в модела са заложени предварително (възможно най-пълно) специфични знания за решаването на конкретната задача. Този подход всъщност не разчита на същинското обучение, т.е. на способността на модела да придобива нови знания. Пример за използването на този подход е системата Meta-DENDRAL [8].

При решаването на задачата за разпознаване на геометрични фигури ние използувахме третия подход. Както бе споменато по-горе този подход не е приложим в по-сложните случаи. Още повече, предложеният формализъм принадлежи към моделите за разпределена обработка на информацията (конекционизъм), при които основното ударение винаги се поставя върху методите за обучение чрез самоорганизация.

При разработката на методи за обучение в езика на мрежовите клаузи изхождаме от конекционисткия модел на обучение. Предложеният подход обаче е доста различен от повечето от методите за обучение на конекционистки мрежи,

където всички потенциални връзки са зададени предварително и чрез множество обучаващи примери се усилват или отслабват. Типично за мрежовите клаузи е създаването на нови връзки. Един метод за обучение със създаване на нови връзки и възли за конекционистки мрежи (recruitment learning) е предложен в [16]. Този метод има известна идейна връзка с обучението в мрежовите клаузи. Това е наличието на т.нар. свободни възли (free units), които в процеса на обучението се свързват с останалата част от мрежата. Обучението при мрежовите клаузи може да се нарече "хибридно", тъй като при създаването на връзки в мрежата се използуват и някои от символните методи на последните два общи подхода към обучението.

В машинното обучение се разглеждат множество стратегии за решаване на задачата. Най-често използувана от тях е *обучение чрез примери* (learning from examples) [17]. Тази стратегия се основава на т.нар. *обобщение* (generalization), срещано главно в две форми: екземпляр-клас (instance-to-class generalization) и част-цяло (part-to-whole generalization). При решаване на задачата за обучение на мрежови клаузи използуваме вариант на *обучение от примери чрез обобщение от типа екземпляр-клас.*

За да покажем възможностите на езика на мрежовите клаузи за обучение, ще разглеждаме следната задача: *при зададена конкретна геометрична фигура чрез набор от страни (структури от типа* ***edge(M,N,S,L)****) и клас, към който тя принадлежи, да се формира мрежова клауза, която да разпознава този клас от фигури.*

В следващите два раздела се описват две възможни решения на поставената задача. Първото е класическо, на базата на стандартния Пролог, докато второто е реализирано чрез възможностите на езика на мрежовите клаузи за динамична промяна на структурата на мрежата. В третия раздел се разглежда една интерпретация на предложения метод за обучение в контекста на индуктивното обучение.

6.1. Обучение чрез Пролог

В раздел 5.1 беше дефинирана операция за получаване на описанието на клас от многостени на базата на конкретен екземпляр от класа. Тази операция се състои в замяната на еднаквите терми в описанието на екземпляра с еднакви (съвместени) променливи. Основната идея в случая е освобождаването на описанието на обекта от конкретни параметри, които нямат отношение към основните свойства, определящи класа, към който той принадлежи, т.е. освобождаване от имената на върховете и стойностите на атрибутите на ръбовете при запазване на свойството еднаквост и различие между тях. Това е конкретната семантика, която се влага в термина *обобщение* (generalization), който използуваме по-нататък при решаването на поставената задача за обучение при разпознаване на геометрични фигури.

Очевидно е, че операцията обобщение, работеща върху списъчно представяне на многостени е лесно реализуема на Пролог. Това е така, тъй като свойството, което се запазва, еднаквост/различие на обекти, е в основата на унификацията. Основата на реализацията на обобщението е един прост предикат, чиято дефиниция е следната:

```
gen([],[],[]):-!.
gen([X|T],[Y|V],[Y|R]):-del(X,T,P,Y,V,Q),!,gen(P,Q,R).
gen([_|T],[_|V],R):-gen(T,V,R).

del(_,[],[],_,[],[]).
del(X,[X|L],M,Y,[Y|P],N):-!,del(X,L,M,Y,P,N).
del(X,[Y|P],[Y|Q],W,[Z|A],[Z|B]):-del(X,P,Q,W,A,B).
```

Предикатът gen преобразува списък от терми в списък от променливи, като унифицируемите терми са заменени със съвместени променливи. Като трети аргумент той връща списък от уникалните променливи, които участвуват в списъка със съвместени променливи. Ето няколко примера за неговата работа:

```
?- gen([a,b,c,a,b],X,Y).
X=[_1,_2,_3,_1,_2]
Y=[_1,_2,_3]

?- gen([a(X),b(b),a(a)],L1,L2).
X=a
L1=[_1,_2,_1]
L2=[_1,_2]
```

На базата на предиката **gen** може да се дефинира и предикат, който преобразува многостен, зададен като списък, в мрежова клауза. Това е предикатът **generalize**, чиято дефиниция следва по-долу. За краткост в списъците, представящи многостените, са пропуснати атрибутите за наклон и дължина.

```
generalize(EdgeList,Net,Name):-single_list(EdgeList,List),
                               gen(List,VarList,Class),
                               single_list(NetList,VarList),
                               make_node(Class,Node,Name),
                               net_clause(NetList,Net,Node).

single_list([],[]):-!.
single_list([edge(X,Y)|T],[X,Y|V]):-single_list(T,V).

make_node(Class,Node,Name):-length(Class,N),M is N+2,
                            functor(Node,node,M),
                            copy_args(Class,Node,1),
                            N1 is N+1,arg(N1,Node,N),
                            arg(M,Node,write(Name)).

copy_args([],_,_):-!.
copy_args([X|T],S,N):-arg(N,S,X),M is N+1,copy_args(T,S,M).

net_clause([],N,N):-!.
net_clause([X|T],(X:R),N):-net_clause(T,R,N).
```

Предикатът може да се използува по следния начин:

```
?- generalize([edge(1,2),edge(2,3),edge(3,1)],X,триъгълник),assert(X).
X=edge(_1,_2):
  edge(_2,_3):
  edge(_3,_1):
  node(_1,_2,_3,3,write(триъгълник))
```

След изпълнението на горната конюнкция в базата от данни е въведена съответната мрежова клауза и следният въпрос предизвиква нейната активация, т.е. отговор на въпроса към кой клас принадлежи въведената фигура.

```
?- edge(a,b),edge(b,c),edge(c,a).
триъгълник
```

Описаното решение на задачата, поставена в началото на настоящия глава е само частен случай на обучение, което независимо, че работи с мрежови клаузи, е еквивалентен на случая със списъчно представяне. Проблемът е, че всеки клас се представя с отделна мрежа (генерирана чрез предиката **generalize**), което означава, че трудностите при разпознаването на повече от един клас, характерни за списъчното представяне остават. С други думи мрежовата клауза осъществява унификацията на конкретен обект с конкретен клас, т.е. изпълнява ролята на предиката **match** (раздел 5.1). Наличието на много класове изисква последователно сканиране на всеки един, което разбира се е далеч от идеята за разпределена обработка. Решението на задачата за разпознаване на много класове беше илюстрирано в раздел 5.3.1 (фигура 3), където мрежата има една обща структура, представяща едновременно всички класове. Именно създаването на такава мрежа чрез обучение е същността на метода за обучение при мрежовите клаузи, описан в следващия раздел.

6.2. Обучение на мрежови клаузи

Както беше споменато в началото на главата обучението на мрежовите клаузи е

частен случай на обучение чрез "самоорганизация" (подхода на конекционистките мрежи). Решаването на поставената задача за обучение при разпознаване на многостени, в съответствие с дефиницията на този вид обучение, предполага решаването на следните задачи:

1. *Създаване на достатъчно гъвкава среда, която да се обучава*. Такава среда може да бъде мрежова клауза, структурирана като набор от свободни и процедурни възли без връзки между тях, т.е. с уникални мрежови променливи, представящи върховете и атрибутите на геометричните фигури. Това е всъщност набор от свободни отсечки и семантични възли, представящи класове фигури. Например:

```
edge(_1,_2,_3,_4):
edge(_5,_6,_7,_8):
edge(_9,_10,_11,_12):
edge(_13,_14,_15,_16):
edge(_17,_18,_19,_20):
edge(_21,_22,_23,_24):
edge(_25,_26,_27,_28):
node(_29,_30,_31,_32,4,fig(ромб)):
node(_33,_34,_35,_36,4,fig(успоредник)):
fig(_37).
```

2. *Процедура, която да променя средата* в съответствие с някакъв обучаващ алгоритъм на базата на въведени конкретни обекти. Промяната на средата в случая означава създаване на няколко типа връзки в мрежата (съвместяване на мрежови променливи) и генериране на нови възли:

а) създаване на топологични връзки. Това става чрез съвместяване на променливите, представящи върховете в свободните възли **edge** (първите две променливи), т.е. сглобяване на фигура от свободни отсечки;

б) налагане на ограничения върху атрибутите (определяне на свойствата успоредност и еднаквост на дължината на ръбовете). Това става чрез съвместяване на променливите в свободните възли **edge**, представящи дължината и

наклона на страните (третата и четвърта променлива);

в) създаване на връзки "част-обект". Това става чрез съвместяване на променливи (върховете на фигурите), участвуващи в свободните възли **edge** и процедурните възли **node**.

За решаване на гореизброените задачи се използува метапроцедурата **gen(<Маркер>)** (раздел 2.7). Изпълнението на следната конюнкция предизвиква структуриране на мрежовата клауза по такъв начин, че тя да може да разпознава конкретни екземпляри на фигурата ромб.

```
?- top(N),
    edge(a,b,0,20),edge(b,c,45,20),edge(c,d,0,20),edge(d,a,45,20),        (1)
    node(a,b,c,d,4,fig(ромб)),
   gen(N).
```

Последователността от ръбове в (1) задава един конкретен пример на ромб, а целта **node** определя семантичния възел, който трябва да се активира при унификация на страните му. Метапроцедурите **top** и **gen** определят областта на действие на процедурата за обобщение (в случая цялата конюнкция). След изпълнение на конюнкцията (1) мрежовата клауза е вече структурирана по следния начин:

```
edge(_1,_2,_3,_4):
edge(_2,_5,_6,_4):
edge(_5,_7,_3,_4):
edge(_7,_1,_6,_4):
edge(_8,_9,_10,_11):
edge(_12,_13,_14,_15):
node(_1,_2,_5,_7,4,fig(ромб)):
node(_16,_17,_18,_19,4,fig(успоредник)).
```

Горната мрежова клауза съдържа описанието на класа на ромбовете и може да разпознава дали дадена геометрична фигура е ромб или не. Например:

```
?- edge(1,2,0,20),edge(2,3,50,20),edge(3,4,0,20),edge(4,1,50,20),fig(X).    (2)
X=ромб
```

Чрез въвеждане на пример на друга фигура мрежата може да се структурира по аналогичен начин за разпознаване на съответния клас. Например следната конюнкция обучава мрежовата клауза за разпознаване на успоредници:

```
?- top(N),
     edge(a,b,0,20),edge(b,c,45,30),edge(c,d,0,20),edge(d,a,45,30),          (3)
     node(a,b,c,d,4,fig(успоредник)),
   gen(N).
```

По такъв начин получаваме мрежова клауза, която е еквивалентна на част от мрежовата клауза (възлите, представящи успоредник и ромб) от примера в раздел 5.3.1:

```
edge(_1,_2,_3,_4):
edge(_2,_5,_6,_4):
edge(_5,_7,_3,_4):
edge(_7,_1,_6,_4):
edge(_2,_8,_9,_10):
edge(_8,_11,_3,_4):
edge(_11,_1,_9,_10):
node(_1,_2,_5,_7,4,fig(ромб)):
node(_1,_2,_8,_11,4,fig(успоредник)):
fig(_12).
```

Чрез повторение на описаната процедура и при наличието на достатъчен брой възли мрежовата клауза може да се обучи да разпознава произволен брой многостени. Важно е да се отбележи, че получената мрежа е винаги *минимална*, т.е. общите части на фигурите не се повтарят в описанието им, в резултат на което разпознаването на обектите става в известен смисъл паралелно (при последователно задаване на входовете). Това се дължи на използуването на

стратегията за последователно търсене на алтернативни свързвания на входовете на мрежата чрез възлите **edge**. При опит за унификация на нов обект, той може да се унифицира изцяло (ако обектът е екземпляр от класа) или частично с класа, записан в мрежовата клауза. Обучението се осъществява при частична унификация или липса на унификация. В този случай част от ръбовете на обекта се унифицират с ръбове на съществуващи класове. Останалите ръбове се унифицират със свободния набор от ръбове, които следват в мрежовата клауза. Всички свързани променливи, независимо дали са от съществуващи класове или от свободни ръбове, се записват в стека на свързванията, където се обработват от процедурата за обобщение **gen**.

Разгледаните дотук мрежи са двуслойни, т.е. между всеки вход и изход има не повече от един процедурен възел. Тази схема на обучение може да се запише по-общо по следния начин:

?- top(M),<данни>,node(t_1,...,t_n,n,<решение>),gen(M)., (4)

където <данни> и <решение> са конюнкции от основни литерали (литерали, съдържащи основни терми), а t_1,...,t_n са основни терми, които се срещат като аргументи на литералите. (В случая приемаме, че в базата от данни се съдържат достатъчно количество възли, които да се използуват като среда за обучение.) Конюнкцията (4) всъщност представлява пример на правило за извод, воден от данните:

<данни> ==> <решение> (5)

Семантиката на това правило е, че ако <данни> е "вярно" (изводимо в логически смисъл), то <решение> също е "вярно". Прилагайки обобщение към правилото (5), термите t_1,...,t_n се заменят с променливи. По такъв начин областта на приложимост на правилото се разширява до всички литерали, които са частни примери (ground instances) на литералите, срещащи се в <данни>.

Въпросите (1) и (3) са примери как да се създават чрез обучение еднослойни мрежови клаузи чрез операцията обобщение, приложена към правила от типа (5). Тази техника може да се използува също и при създаване на многослойни мрежи. Това може да стане стъпково, прилагайки въпроса (4) за всеки слой от мрежата, като по този начин се задава един пример на верига от правила да извод, воден от данните:

<данни> ==> <данни$_1$>
<данни>,<данни$_1$> ==> <данни$_2$>
...
<данни>,<данни$_1$>,...,<данни$_m$> ==> <решение>

6.3. Индуктивно обучение на понятия

Машинното обучение е един от трудните проблеми в ИИ. Повечето успехи в това направление са постигнати в тесни и добре формализирани области. Дори и в такива области обаче съществуват редица трудности. Да разгледаме например задачата за *обучение на понятия*, която е частен случай на *индуктивно обучение чрез примери*. Тази задача в логически контекст се дефинира по следния начин [23]: *понятието* е предикат. *Описанието на понятието* е множество от клаузи, които обясняват (извеждат) съответния предикат в логически смисъл. *При дадено множество от основни примери (ground instances) на даден предикат, трябва да се намери описанието му.* Практическите системи за обучение на понятия въвеждат редица ограничения върху описаната обща задача. Повечето от тях засягат описанието на обучавания предикат. Езикът, с който той се описва, често се ограничава до *дефинитни клаузи* (клаузи без отрицателни цели тялото) [65,66]. Въвежда се и т.нар. *схема* (language scheme или bias) [76], която се използува да ограничи още повече езика на описанието, като по този начин се намали неограничения брой възможни хипотези за описанието на обучавания предикат. Използуването на дефинитни клаузи решава също *проблема с отрицанието*. В

повечето системи обяснението на понятието се базира на дедукция, при която отрицанието се дефинира като неуспешен извод (negation by failure), т.е. използува се *предположението за затворения свят* (Closed World Assumption - CWA). Последното обаче е в противоречие с основната идея на обучението, т.е. способността на системата да придобива нови знания предполага отвореност на света. Освен избягване на отрицанието съществуват и други подходи, като например описаният в [15], където се използуват явно отрицание на предикатите и тризначна логика.

Повечето методи за обучение на понятия се базират на класически дедуктивни системи за извод (например Пролог). Такива са системите, описани в [15,65,66]. В тези случаи процесът на обучение е съществено различен от дедуктивния процес, използуван при обяснението (извеждането) на обучените понятия. Когато се генерира хипотеза се извършва глобално претърсване на базата от клаузи, което понякога предизвиква комбинаторна експлозия. В Пролог например извеждането на понятие се осъществява чрез ограничено и насочено търсене (извод воден от целта), а индуктивното извеждане на хипотеза става чрез обработка на всички положителни литерали в базата от данни (факти или глави на клаузи). Последното може да се разглежда като извод, воден от данните. Следователно основната идея е *да се реализират методите за обучение на понятия в дедуктивна система, базирана на извод, воден от данните*. За тази цел успешно може да се използува езикът на мрежовите клаузи. Всъщност описаната в предишния раздел процедура за обучение на мрежови клаузи може естествено да се интерпретира като решение на задачата за индуктивно обучение на понятия. За тази цел ще дефинираме строго задачата за обучение на понятия в контекста на езика на мрежовите клаузи. В дефиницията се използува понятието *правило за извод, воден от данните*. Това е мрежова клауза, съставена от един процедурен и няколко свободни възли, свързани със съвместени променливи. (Такова правило е например мрежовата клауза, която се получава при преобразуването на програмна клауза по правилата, описани в началото на раздел 3.1.)

Дефиниция. Задача за обучение на понятия в езика на мрежовите клаузи:

Дадено:

1. Основни знания (background knowledge) - множество от правила за извод, воден от данните.
2. Обучаваща се среда - достатъчно богато множество от свободни възли с уникални мрежови променливи.
3. Пример на предикат - основен литерал P.
4. Положителни данни - основни литерали $PD_1,\ldots,PD_m$.
5. Отрицателни данни - основни литерали $ND_1,\ldots,ND_n$.

Да се намери:

Разширени основни знания, така че P се извежда чрез извод, воден от данните, от основните знания и положителните данни, но не може да се изведе от основните знания и отрицателните данни.

Важно е да се отбележи, че по отношение на стандартната дефиниция в [23], ролите на предикатите и данните са разменени. Това може да се обоснове с факта, че в езика на мрежовите клаузи данните, а не предикатите са инвариантната част от знанията, която подлежи на обобщение чрез индукция.

Решението на поставената задача се базира само на операцията обобщение, реализирана чрез метапроцедурата gen. За тази цел се разглеждат следните две процедури:

1. *Обработка на положителните данни.* Изпълнява се въпросът ?-$PD_1,\ldots,PD_m$. След това се изпълнява операцията обобщение (gen) и към основните знания се добавя процедурният възел $node(X_1,\ldots,X_k,k,P')$. $X_1,\ldots,X_k$ са всичките различни променливи, участвуващи в операцията обобщение, а P' е обобщението на P, в което уникалните аргументи са запазени.

2. *Обработка на отрицателните данни.* Изпълнява се въпросът ?-$ND_1,\ldots,ND_n$. След това се извършва операцията обобщение и участвуващите в нея различни променливи $(Y_1,\ldots,Y_q)$ се добавят като подтискащи връзки към съответния процедурен възел, т.е. получава се възелът $node(X_1,\ldots,X_k,\sim Y_1,\ldots,\sim Y_q,k,P')$.

Важно е да се отбележи, че променливите, участвуващи в операцията обобщение, не са само променливите, участвуващи в свободните възли на обучаващата се среда, директно унифицирали се с данните. Тъй като данните могат да активират процедурни възли от основните знания, допълнителни променливи също могат да участвуват в генерираната хипотеза. Това е случаят, когато се обучават йерархични предикати. Тук е мястото да се подчертае, че *в процеса на индуктивно обучение се използува същия извод както и при обяснението (извеждането) на предикатите - извод, воден от данните.* В този смисъл може да се твърди, че обучението в езика на мрежовите клаузи е естествено свързано с основните механизми на езика.

Описаната схема за решаване на задачата за обучение на понятия има следните специфични особености:

1. Тъй като различните правила за извод могат да имат съвместени променливи, основните знания на системата не са конюнкция от клаузи на Хорн, а специален вид формула от предикатното смятане (дефинирана в раздел 3.3). В този смисъл при всяка стъпка на индукцията текущата формула (основните знания) се разширява с нов конюнкт (правило за извод, воден от данните).

2. Всяка новосъздадена хипотеза (правило за извод) се добавя към съществуващите основни знания. По-този начин обучаващата се среда се структурира и превръща в основни знания. Тъй като отделните правила за извод могат да имат съвместени променливи, съществуващите основни знания могат де се променят. Това може да се случи, ако нова хипотеза съвмести променливи, принадлежащи към основните знания. За да се избегнат подобни ефекти (*немонотонност на индуктивния извод*) трябва да се следва конкретна *стратегия за обучение.* Най-общо съществуват две стратегии за обучение:

а) *използуване само на положителни данни.* В този случай данните трябва да се задават в специален ред - от по-конкретните към по-общите. Това означава, че по време на обучението в основните знания не трябва да има правило, което поглъща (subsumes) текущо извежданото правило. Това условие е спазено в описания в предишния раздел пример на обучение за разпознаване на

геометрични фигури. Тъй като правилото за извод на успоредник поглъща правилото за извод на ромб (ромба е частен случай на успоредник), то обучаващата последователност е първо ромб, а след това успоредник. В противен случай примерът на ромба би унифицирал правилото за извод на успоредник и при прилагане на обобщението някои от променливите, задаващи ограниченията на успоредника, ще се съвместят за да отразят ограниченията на ромба. По тази начин правилото на ромба ще припокрие правилото на успоредника, което разбира се ще направи некоректни основните знания на системата.

б) *използуване на положителни и отрицателни данни.* В този случай данните могат да се обработват в произволен ред. След всяка стъпка на индукцията обаче трябва да се извършва актуализация на основните знания. Това става като положителните данни на последната изведена хипотеза трябва да се използуват като отрицателни такива за модифицирането (чрез описаната по-горе процедура 2) на всички грешно активирали се правила за извод от основните знания.

3. Основните знания са *минимални* в смисъл, че всички свободни възли, принадлежащи едновременно към няколко правила за извод се срещат само веднъж в основните знания. Това свойство беше обяснено в предишния раздел чрез примера за обучение на геометрични фигури.

6.4. Резюме

Обучението е задължителен атрибут на конекционистките модели. В настоящата глава е описана схема за обучение в езика на мрежовите клаузи, която допълва и доразвива конекционистките характеристики на езика. Схемата е описана чрез решаване на задачата за създаване на мрежа, разпознаваща геометрични фигури. Описани са решения чрез стандартен Пролог и чрез езика на мрежовите клаузи. Показано е, че обучението в езика на мрежовите клаузи е естествено свързано с основния механизъм да извод, воден от данните. Дадена е формална интерпретация на механизма за обучение в мрежовите клаузи в рамките на *индуктивното обучение*

на понятия (concept learning). Подчертани са предимствата на описания подход, по отношение на класическите обучаващи се системи, базирани на дедукция с воден от целта извод.

ГЛАВА 7. Реализация на езика на мрежовите клаузи

В тази глава се прави кратко описание на реализацията на езика на мрежовите клаузи. Целта на изложението е да се направи сравнение с Пролог и да се подчертаят сравнително неголемите реализационни усилия на фона на постигането на значителна експресивност на езика.

Езикът на мрежовите клаузи е интегриран в изчислителната среда на Пролог на ниво реализационен език (Паскал). Важно е да се отбележи, че при реализацията на езика се използуват в тяхната оригинална форма почти всички основни механизми на Пролог, като унификацията, алгоритъма за търсене с възврат, достъпа и модификацията на базата от данни. Единственото разширение, което всъщност прави новия език, засяга променливите. Това е напълно естествено, имайки предвид основната концепция за управлението в езика на мрежовите клаузи. И двата управляващи механизма се основават на унификацията. Докато обаче в Пролог управлението се задава явно чрез реда на клаузите и целите в тях, при мрежовите клаузи управлението е водено от данните. И тъй като данните са терми, разпространявани от мрежовите променливи, то ясно е, че мрежовите променливи трябва да осъществяват изцяло управлението в езика.

Променливата в Пролог е пасивен елемент на езика, реализиран като свободен указател към поле, съдържащо самите данни (терми). Освен основната си роля на указател към терм мрежовата променлива е активен елемент в управлението на езика на мрежовите клаузи. За да изпълнява тази функция, реализацията на мрежовата променлива има две допълнителни особености:

1. Мрежовата променлива не се копира, когато при успешна унификация със съответната цел се активира нова клауза. Всъщност това не е добавяне на ново качество, а по-скоро премахване на характерна особеност на променливите в Пролог. По този начин мрежовата променлива е *глобална променлива*, директно достъпна от всички цели, които унифицират клауза, в която тя се появява.

2. При реализацията на мрежовата променлива се използуват три допълнителни указателя, съответно за трите типа процедурни възли. При

зареждане на мрежовата програма тези указатели се установяват със стойности, сочещи към съответните процедурни възли, в които мрежовата променлива се среща. При унификация на мрежовата променлива указателите определят кои процедурни възли трябва да се активират (или съответните им условия да се проверят).

Реализацията на мрежовите променливи с отлагане на унификацията е по-сложно, но то също не засяга другите оригинални механизми на Пролог, които се използуват при създаването на езика на мрежовите клаузи. Поведението на програмите, използуващи унификация с отлагане, се определя единствено от функционалните качества на променливите с отлагане, т.е. от факта, че те не се свързват (тип 1) или се свързват при определени условия (тип 2), което от своя страна се отчита от стандартната процедура за унификация. Някои от принципите, използувани при реализацията на променливите с отлагане, са описани в раздел 2.8.

Останалите елементи на езика на мрежовите клаузи са стандартни конструкции от езика Пролог, които получават автоматически новите си свойства само при използуването на мрежови променливи в тях. Така например процедурните и свободни възли в езика са факти на Пролог с мрежови променливи в тях. Те могат да се обработват по стандартния начин чрез вградените предикати на Пролог за достъп до базата от данни. Единствената синтактична разлика от Пролог е, че мрежовите клаузи дефинират специална област в базата от данни, в която важат две условия: променливите са мрежови и техните имена са глобални.

Езикът на мрежовите клаузи е реализиран като разширение на версията КСИ-ПРОЛОГ [1,2], работеща върху персонални компютри IBM-PC. Всички възможности на тази версия са достъпни и в новата разширена версия. Това дава възможност за използуване на процедури на Пролог за символна обработка в комбинация с разпределени мрежови модели, реализирани на езика на мрежовите клаузи, т.е. практическа интеграция на символни и конекционистки модели в единна програмна среда.

ЗАКЛЮЧЕНИЕ

В настоящата работата са решени следните основни задачи:

1. Направен е преглед на подходите за създаване на *паралелен Пролог*, като са анализирани теоретичните и практически трудности при реализацията им, които мотивират създаването на езика на мрежовите клаузи.

2. Създаден е език за разпределена обработка, наречен *език на мрежовите клаузи*, чието основно предназначение е създаването на модели със *разпределено изчисление без наличието на централизирано управление на базата на унификацията*.

3. Разработена е схема за използуване на езика на мрежовите клаузи за *дедуктивен извод*. Дефинирано е съответствие между мрежовите клаузи и клаузите на Хорн, на базата на което схемата за разпространяваща се активация реализира *логически извод, воден от данните*. Този извод може да се прилага към клаузи на Хорн, както и към по-широк клас формули на предикатното смятане от първи ред. Показани са предимствата на езика, по отношение на използуването му като дедуктивна система.

4. Разработена е схема за реализация на *извод по премълчаване* (default reasoning) чрез езика на мрежовите клаузи. Предложената схема попада в общата схема на *присвояване на стойности по премълчаване на променливи* (default assignment to variables), като в същото време показва някои значителни предимства, най-важното от които е възможността за йерархична организация на извода по премълчаване.

5. Решена е задачата за представяне и разпознаване на многостени. Предложени са две решения - чрез стандартния Пролог и чрез езика на мрежовите клаузи. Второто решение показва предимствата на езика на мрежовите клаузи при решаването на задачи за структурно представяне на обекти, частен случай на които е разгледаната задача, както и доказва възможностите на езика за създаване на модели за разпределена обработка на информацията (конекционистки модели).

6. Разработена е схема за обучение чрез езика на мрежовите клаузи, която допълва и доразвива конекционистките му характеристики. Схемата е описана чрез решаване на задачата за създаване на мрежа, разпознаваща геометрични фигури. Описани са решения чрез стандартен Пролог и чрез езика на мрежовите клаузи. Показано е, че обучението в езика на мрежовите клаузи е естествено свързано с основния механизъм да извод, воден от данните. Дадена е формална интерпретация на механизма за обучение в мрежовите клаузи в рамките на *индуктивното обучение на понятия*, която показва предимствата на описания подход по отношение на класическите обучаващи се системи, използуващи дедукция с воден от целта извод.

7. Езикът на мрежовите клаузи е реализиран като разширение на версията КСИ-ПРОЛОГ, работеща върху персонални компютри от типа IBM-PC.

Резултатите, описани в настоящата работа не изчерпват всички аспекти на езика на мрежовите клаузи, като и всички негови възможни приложения. Тъй като той интегрира два съществено различни подхода към решаването на задачите на ИИ - символния и конекционисткия, очевидно е необходимо по-нататъшно изследване на възможностите му, особено в областите, в които всеки един отделен подход е неефективен. Следните задачи за по-нататъшна работа могат да се разглеждат като непосредствени следствия от постигнатите резултати:

- Изследване на възможностите за паралелна реализация на езика на мрежовите клаузи. Схемата за разпространяваща се активация може директно да се използува за паралелна реализация. За тази цел е необходимо да се организира паралелната работа на процедурите във възлите на мрежата като отделни процеси. Тогава съществуващите в езика условия за активация на процедурите ще осъществяват синхронизацията между процесите.

- Формално разглеждане на механизма за извод по премълчаване в рамките на немонотонните логики. Това от една страна би дало още една формална семантика на езика на мрежовите клаузи, а от друга - практическа реализация на априори трудните проблеми за реализация на немонотонните логики.

- Използуване на езика на мрежовите клаузи за програмиране с

ограничения. *Програмирането с ограничения* (constraint satisfaction) е мощен метод за решаване на задачи от областта на обработката на изображения и редица други области в ИИ. Реализации на този метод се правят както в логическото програмиране така и чрез конекционистки модели. Особено популярен е подходът на логическото програмиране, който доведе до обособяването на отделна област, наречена *логическо програмиране с ограничения* (constraint logic programming). Задачата за разпознаване на многостени често се решава чрез програмиране с ограничения. Описаното решение на езика на мрежовите клаузи също показва някои от особеностите на този вид задачи. В този смисъл езикът на мрежовите клаузи е интересен (и вероятно плодотворен) подход за решаване на задачите за програмиране с ограничения.

- Изследване на възможностите на езика на мрежовите клаузи за реализация на конструктивно отрицание в логическото програмиране.

ЛИТЕРАТУРА

1. Дочев, Д., Х. Дичев, З. Марков, Г. Агре. Програмиране на Пролог - основи и приложения. С., Наука и изкуство, 1989.
2. КСИ-ПРОЛОГ. Ръководство за програмиста. ИТКР-БАН, НИПЛ "Програмно осигуряване", С., 1986.
3. Arbib M.A. Brains, Machines, and Mathematics (second edition). Springer-Verlag, 1987.
4. Bell, J. Nonmonotonic reasoning, nonmonotonic logics and reasoning about change. *Artificial Intelligence Review*, 4 (1990), 79-108.
5. Bic, L. & C. Lee. A Data-driven Model for a Subset of Logic Programming. *ACM Transactions on Programming Languages and Systems*, Vol.9, No.4, 1987, 618-645.
6. Bobrow, D.G. & T. Winograd. An overview of KRL-0, a knowledge representation language. *Cognitive Science*, Vol.1 (1977), No.1.
7. Bratko, I. Prolog programming for Artificial Intelligence. Addison-Wesley, Amsterdam, 1986.
8. Buchanan, B.G, E.A. Feigenbaum. DENDRAL and Meta-DENDRAL: Their Applications Dimension, *Artificial Intelligence*, Vol.11, pp. 5-24, 1978.
9. Chang C.-L., Lee R. C.-T. Symbolic logic and mechanical theorem proving. Academic Press, London, 1973.
10. Chater, N. & M. Oaksford. Autonomy, implementation and cognitive architecture: A replay to Fodor and Pylyshin. *Cognition*, 34 (1990), 93-107.
11. Clark, K.L. & F.G. McCabe. Micro-Prolog: Programming in Logic. Prentice-Hall International, 1984.
12. Clocksin,W.F, C.S.Mellish. Programming in Prolog. Springer-Verlag, 1981.
13. Colmerauer, A., H. Kanoui, P. Roussel and R. Pasero. Un Systeme de Communication Homme-Machine en Francais, Groupe de Recherche en Intelligence Artificielle, Universite d'Aix-Marseille, 1973.
14. Connectionism and information processing abstractions. *AI magazine*, Winter 1988, 25-34.
15. De Raedt, L. and M. Bruynooghe, On Negation and Three-valued Logic in Interactive Concept-Learning, in: *Proceedings of ECAI-90*, Stockholm, Sweden, August 6-10, 1990, pp.207-212.
16. Diederich, J. Connectionist Recruitment Learning. In: *Proceedings of ECAI-88*, Munich, August 1-5, 1988, 351-356.

17. Dietterich, T.G., R.J. Michalski. A Comparative Review of Selected Methods for Learning from Examples. In: *Machine Learning - An Artificial Intelligence approach*, Michalski, R, J. Carbonell and T. Mitchell (Eds.). Tioga Publishing Company, Palo Alto, CA, 1983.
18. Engelbrecht, J., F.Wahl. Polyhedral Object Recognition using Hough-Space Features. IBM Research Report RZ 1486(#54038), IBM Zurich Research Laboratory, 1986.
19. Feldman, J.A. & D.H. Ballard. Connectionist models and their properties. Cognitive science, vol.6, 205-254.
20. Fodor, J.A. & Z.W. Pylyshin. Connectionism and cognitive architecture: A critical analysis. *Cognition*, 28 (1988), 3-71.
21. Friedman, D.P. and D.S.Wise. CONS should not evaluate its arguments. In Automata, Languages and Programming, S. Michaelson and R. Milner (Eds.), Edinburgh University Press, pp.257-284.
22. Garey, M., D. Johnson. Computers and Intractability - A Guide to the Theory of NP-Completeness. Freeman, San Francisco, 1979.
23. Genesereth, M.R, N.J.Nilsson, *Logical foundations of Artificial Intelligence*, Morgan Kaufmann, Los Altos, 1987.
24. Gregory, S. Parallel Logic Programming in PARLOG. Addison-Wesley, 1987, p.217.
25. Guerrieri E., V. Grover. Octree Solid Modeling with PROLOG. In: *Application of AI in Engineering Problems*, Sriram D., R. Adey (Eds.), vol. 2. Springer-Verlag, 1986.
26. Henderson, P. and J.H.Morris. A lazy evaluator. In Proceedings of the 3rd ACM Symposium on Principles of Programming Languages, 1976, ACM, pp.95-103.
27. Jorrand, Ph. Design and implementation of a parallel inference machine for first order logic, in: *Proc. of PARLE Conference* (Lecture Notes in Computer Science No. 259, Springer-Verlag, 1987).
28. Jorrand, Ph. Design and implementation of a parallel inference machine for first order logic. In: *Procedeengs of PARLE Conference* (Lecture Notes in Computer Science No. 259, Springer-Verlag, 1987).
29. Jorrand, Ph. Parallel Inference Machine for First Order Logic: A Rigorous Approach to the Design, A New Paradigm for "Computing". First Franco-Bulgarian Workshop on LOGIC PROGRAMMING and AUTOMATED INFERENCING, 1988, LIFIA-Grenoble.
30. Jorrand, Ph. Process and Term Algerbras for Abstract Specification of Parallel Inference in Horn Clause Logic. AIICSR, Vysoke Tatry, November 6-10, 1989.

31. Jorrand, Ph. Term Rewriting as a Basis for the Design of a Functional and Parallel Programming Language. A case study: the language FP2. In: Lecture Notes in Computer Science 232 (eds. W.Bibel and Ph.Jorrand). Springer-Verlag, 1986, pp.221-276.

32. Kaplan R.M. & J.T.Maxwell III, 1988. Constituent coordination in Lexical-functional grammar. In: *Proceedings of COLING'88*, pp.303-305.

33. Kaplan, S, M. Weaver & R. French. Active Symbols and Internal Models: Towards a Cognitive Connectinism. *AI & Society*, 4 (1990), 51-71.

34. Kodratoff, Yves, Jean-Gabriel Ganascia. Improving the generalization step in learning. In: *Machine Learning - An Artificial Intelligence approach*, Volume II (eds. Michalski, R, J. Carbonell and T. Mitchell). Morgan Kaufmann Publishers, 1986.

35. Kowalski, R.A. Logic for Problem Solving. Elsevier North-Holland, New York, 1979.

36. Lloyd J. W. Foundations of Logic Programming. Springer-Verlag, 1984.

37. Markov Z., Th. Risse. Prolog Based Graph Representation of Polyhedra, in: *Proceedings of AIMSA'88* (Artificial Intelligence III, North-Holland, Amsterdam, 1988), 187-194.

38. Markov, Z and Ch. Dichev, Logical inference in a network environment, in: *Proceedings of AIMSA'90* (Artificial Intelligence IV, North-Holland, Amsterdam, 1990), 169-178.

39. Markov, Z. A framework for network modeling in Prolog, in: *Proceedings of IJCAI-89*, Detroit, U.S.A (1989), 78-83.

40. Markov, Z., C. Dichev and L. Sinapova, The Net-Clause Language - a tool for describing network models, in: *Proceedings of the Eighth Canadian Conference on AI*, Ottawa, Canada, 23-25 May, 1990, 33-39.

41. Markov, Z., L. Sinapova and Ch. Dichev. Default reasoning in a network environment, in: *Proceedings of ECAI-90*, Stockholm, Sweden, August 6-10, 1990, 431-436.

42. McCulloch, W.S & W. Pitts. A logical calculus of the ideas immanent in nervious activity. *Bulletin of Mathematical Biophisics*, 5 (1943), 115-133.

43. Michalski, R, J. Carbonell and T. Mitchell (eds.). Machine Learning - An Artificial Intelligence approach. Tioga Publishing Company, Palo Alto, CA, 1983.

44. Michalski, R. A Theory and Methodology of Inductive Learning. In: *Machine Learning - An Artificial Intelligence approach*, Michalski, R, J. Carbonell and T. Mitchell (Eds.). Tioga Publishing Company, Palo Alto, CA, 1983.

45. Michalski, R. Understanding the nature of learning. In: *Machine Learning - An Artificial Intelligence approach*, Volume II (eds. Michalski, R, J. Carbonell and T. Mitchell). Morgan Kaufmann Publishers, 1986.

46. Minsky, M & S. Papert. Perceptrons, MIT Press, Cambridge, MA, USA, 1969.
47. Moss, Chris. Artificial Intelligence and Symbols. *AI & Society*, 3 (1989), 345-356.
48. Murray, N.V. Completely non-clausal theorem proving, *Artificial intelligence* 18 (1982), 67-85.
49. Naish, L., (Ed.). MU-Prolog 3.1db Reference Manual. Dept. of Computer Science, University of Melbourne, 1984, p. 247.
50. Nilsson, N.J. Learning Machines. McGraw-Hill, New York, 1965.
51. Pollack, J.B. Connectionism: past, present, and future. *Artificial Intelligence Review* 3 (1989), 3-20.
52. Quinlan, J.R. Discovering Rules from Large Collections of Examples: A Case Study. In: *Expert Systems in the Microelectonics Age*, D. Michie (Ed.), Edinburg University Press, Edinbugrh, 1979.
53. Quintus Prolog User's Guide and Reference Manual. Quintus Computer Systems, Palo Alto, 1986.
54. Reiter R., A logic for default reasoning, Artificial Intelligence, vol.13 (1980), pp. 81-132.
55. Reiter, R. On Closed World Data Bases. In: *Logic and Databases*, Gallaire, H. and J. Minker (Eds.), Plenum Press, New York, 1978.
56. Reiter, R., On Resoning by Default. In Proceedings of TINLAP-2, Theoretical Issues in natural Language Processing-2, University of Illinois at Urbana-Champaign, 1978, pp.210-218.
57. Roberts, R.B. & I. Goldstein. The FRL Manual, A.I. Memo No.409, M.I.T., Sept. 1977.
58. Robinson, J.A. A Machine-oriented Logic Based on the Resolution Principle, JACM,12,1(Jan.1965),23-41.
59. Rosenblatt, F. The Perceptron: A Probabilistic Model for Information Storage and Organization in the Brain. *Psychological Review*, Vol.65, 386-407, 1958.
60. Rumelhart, D.E, J.L. McClelland & the PDP research Group (Eds.). Parallel Distributed Processing: Explorations in the in the microstructure of cognition, Vol.1, MIT Press, Cambridge, MA, USA, 1986.
61. Rumelhart, D.E, J.L. McClelland & the PDP research Group (Eds.). Parallel Distributed Processing: Explorations in the in the microstructure of cognition, Vol.2: Psychological and biological processes, MIT Press, Cambridge, MA, USA, 1986.
62. Rumelhart, D.E. & D. Zipser. Feature Discovery by Competitive Learning. Cognitive Science, 9 (1985), 75-112.

63. Rumelhart, D.E., G.E. Hinton & R.J. Williams. Learning Internal Representations by Error Propagation. In: *Rumelhart, D.E. & J.L. McClelland (Eds.), Parallel Distributed Processing*, Vol.1: Foundations, The MIT Press, Cambridge, MA, 1986.

64. Shapiro, E. Concurrent PROLOG: A Progress Report. In: Lecture Notes in Computer Science 232 (eds. W.Bibel and Ph. Jorrand). Springer-Verlag, 1986, 277-313.

65. Shapiro, E.Y. Algorithmic Program Debugging (MIT Press, Cambridge, MA, 1983).

66. Shapiro, E.Y. Inductive inference of theories form facts, Tech. Rept.192, Department of Computer Science, Yale University, New Haven, CT (1981).

67. Sinapova, L, A network parsing scheme, in: *Proceedings of AIMSA'90* (Artificial Intelligence IV, North-Holland, Amsterdam, 1990), 383-392.

68. Sterling, L. & E. Shapiro. The Art of Prolog. MIT Press, 1986.

69. Sterling, L. & M. Codish. Pressing for Parallelism: A Prolog Program Made Concurrent. In: *Concurrent prolog: Collected Papers* (Ed. E. Shapiro), Vol.2, MIT Press, 1987, 304-350.

70. Stickel M. E. An Introduction to Automated Deduction. In: Lecture Notes in Computer Science 232 (eds. W.Bibel and Ph.Jorrand). Springer-Verlag, 1986.

71. Trehan, R & P.F. Wilk. Issues of non-determinism in Prolog and the Commited Choice non-deterministic Logic Languages. *Artificial Intelligence Review*, 4 (1990), 211-232.

72. Turner, R. Logics for Artificial Intelligence, Ellis Horwood, Chichester, 1984.

73. Ueda, K. Guarded Horn Clauses. ICOT Technical Report TR-103, 1985.

74. Ueda, K. Making Exhaustive Search Programs Deterministic, Part II. In: *Fourth International Conference on Logic Programming* (Ed. J. Lassez), MIT Press, Melbourne, 1987, 356-375.

75. Ullmann, J. An Algorithm for Subgraph Isomorphism. Journal of the ACM, vol.23, No.1, 1976.

76. Utgoff, P.E., Shift of Bias for Inductive Concept Learning. In: *Machine Learning - An Artificial Intelligence approach*, Michalski, R, J. Carbonell and T. Mitchell (Eds.), Vol. II, Morgan Kaufmann, Los Altos, 1986.

77. Van Emden, M.H and G.J.de Lucena. Predicate Logic as a language for parallel programming. In Logic Programming, K.L.Clark and S.A.Tarnlund (Eds.), Academic Press, London, 1982, pp. 189-198.

78. Warren, D.H.D. Applied logic - its use and implementation as a programming tool. PhD thesis, Department of Artificial Intelligence, University of Edinburgh, 1977.

79. Winston, P.H. Learning Structural Descriptions from Examples. In: *Psychology of Computer Vision*, P.H. Winston (Ed.), McGraw-Hill, New York, 1975.

ПУБЛИКАЦИИ ПО ТЕМАТА НА ДИСЕРТАЦИЯТА

1. Markov Z., Th. Risse. Prolog Based Graph Representation of Polyhedra, in: *Proceedings of AIMSA'88* (Artificial Intelligence III, North-Holland, Amsterdam, 1988), 187-194.
2. Markov, Z. A framework for network modeling in Prolog, in: *Proceedings of IJCAI-89,* Detroit, U.S.A (1989), 78-83.
3. Markov, Z and Ch. Dichev, Logical inference in a network environment, in: *Proceedings of AIMSA'90* (Artificial Intelligence IV, North-Holland, Amsterdam, 1990), 169-178.
4. Markov, Z., C. Dichev and L. Sinapova, The Net-Clause Language - a tool for describing network models, in: *Proceedings of the Eighth Canadian Conference on AI*, Ottawa, Canada, 23-25 May, 1990, 33-39.
5. Markov, Z., L. Sinapova and Ch. Dichev. Default reasoning in a network environment, in: *Proceedings of ECAI-90*, Stockholm, Sweden, August 6-10, 1990, 431-436.